digital

software

RSTS/E Task Builder Reference Manual

Order Number: AA-5072D-TC

August 1990

This document describes the RSTS/E Task Builder (TKB), and tells how you use it to link programs.

Operating System and Version: RSTS/E Version 10.0

Software Version: RSTS/E Version 10.0

digital equipment corporation
maynard, massachusetts

August 1990

Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation. The following are trademarks of Digital Equipment Corporation:

ALL–IN–1	DEUNA	RSX
DEC/CMS	DIBOL	RT
DECdx	EDT	RT-11
DEC/FMS-11	IAS	TOPS-10
DECmail	LA	TOPS-21
DECnet	LN01	ULTRIX
DECnet/E	Micro/RSX	UNIBUS
DECSA	OS/8	VAX
DECserver	PDP	VAXmate
DECsystem-10	PDP-11	VMS
DECSYSTEM-20	PDT	VT
DECUS	Q-BUS	WPS-PLUS
DECworld	RMS-11	Rainbow
DELUA	RSTS	digital™
DEQNA		

IBM is a registered trademark of International Business Machines Corporation.
RMS is a trademark of American Management Systems, Inc.

Contents

Preface xv

Chapter 1 Introduction

1.1 What the Task Builder Does 1–2
1.1.1 Linking 1–2
1.1.2 Overlays 1–3

1.2 Relationship to the DCL LINK Command 1–4

Part I Getting Started

Chapter 2 Building Programs

2.1 Job Area 2–2
2.1.1 Your Program Within the Job Area 2–2

2.2 Libraries 2–3
2.2.1 Disk Libraries 2–4
2.2.2 Resident Libraries 2–5
2.2.3 Comparison of Disk and Resident Libraries 2–7

2.3 How to Run the Task Builder 2–7
2.3.1 Command Line 2–8
2.3.2 Multiline Command 2–9
2.3.3 Options 2–10
2.3.4 The LIBR and RESLIB Options 2–10
2.3.5 The CLSTR Option 2–11

2.4 Examples of Simple Builds 2–14
2.4.1 BASIC-PLUS-2 Examples Including Disk, Resident, and Cluster Libraries 2–15
2.4.2 PDP-11 COBOL Example Including Two Disk Libraries 2–16
2.4.3 COBOL-81 Examples Including Disk Library and Cluster Libraries 2–16
2.4.4 DIBOL Example Including Disk and Resident Libraries 2–16
2.4.5 FORTRAN-77 Examples Including One Disk Library 2–17
2.4.6 MACRO Examples Including Resident Libraries 2–17

Part II Overlays

Chapter 3 The Basic Concepts

3.1 What are Overlays? 3–2

3.2 Constructing an ODL File: .ROOT, .FCTR, and .END Commands 3–3
- 3.2.1 The .ROOT Command 3–3
- 3.2.2 The .FCTR Command 3–4
- 3.2.3 The .END Command 3–5
- 3.2.4 Flexibility of the Overlay Description Language 3–5

3.3 Using an ODL File When You Run TKB 3–6

3.4 The Memory Map File 3–6

3.5 Designing Overlays Intelligently: Considering Space and Time 3–7
- 3.5.1 Considering Space: Two Possibilities for Example 3–8
- 3.5.2 Considering Time: Reducing Disk Access 3–9

3.6 Logical Independence of Items in Overlay Structure 3–10

3.7 Resolution of Global Symbols 3–11
- 3.7.1 What Is a Global Symbol? 3–11
- 3.7.2 Undefined, Multiply Defined, and Ambiguously Defined Global Symbols 3–12
- 3.7.3 How Routines Are Inserted from Libraries 3–13
- 3.7.4 The Default Library 3–15

Chapter 4 Co-Trees: Another Way to Save Space

4.1 The Co-Tree Structure 4–1

4.2 Using the .NAME Command for a Co-Tree Root 4–4

4.3 Designing the Most Space-Saving Co-Trees 4–5

4.4 Co-Trees and High-Level Languages 4–6
- 4.4.1 Sample Source Program and Subprograms 4–6
- 4.4.2 Outlining the Sample Program's Call Structure 4–7
- 4.4.3 Compiling the Sample Program and Subprograms 4–8
- 4.4.4 First Build for Sample Program: Putting Subprograms in the Root 4–9
- 4.4.5 Second Build for Sample Program: Using a Co-Tree 4–10
- 4.4.6 Third Build for Sample Program: Restructured Tree and Library Routines in Root 4–15
- 4.4.7 Further Tips 4–17
- 4.4.8 Using Co-Tree Techniques with the Default Library 4–17

Chapter 5 The Autoload Indicator

5.1 What are Autoload Vectors? 5–1

5.2 Where are Autoload Vectors Really Needed? 5–3

5.3 How to Request Specific Autoload Vectors 5–4
- 5.3.1 Asterisk Before File Names and Program Sections 5–4
- 5.3.2 Asterisk Before Items in Parentheses 5–5
- 5.3.3 Asterisk Before Names Defined in .FCTR Commands 5–5
- 5.3.4 Asterisk Before Names Defined in .NAME Command 5–5

5.4 Example of Specific Autoload Vector Requests 5–6

5.5 If You Make a Mistake 5–6

Chapter 6 Working with Program Sections

6.1 What is a Program Section? 6–1

6.2 Allocating Space for Global Program Sections 6–2

6.3 How the Task Builder Orders Program Sections 6–3

6.4 The Task Builder's .PSECT Command 6–5

6.5 Using .NAME to Make a Data PSECT Autoloadable 6–5

6.6 More About Program Sections: Deciphering the Map 6–6
- 6.6.1 Source for Program USER 6–8
- 6.6.2 Source for Subprogram INTRO 6–8
- 6.6.3 Source for Subprogram CRUNCH 6–8
- 6.6.4 Source for Subprogram CHATR 6–8
- 6.6.5 Overlay Description File FRED.ODL 6–8
- 6.6.6 Task Builder Command File 6–9
- 6.6.7 Task Builder Listing 6–9

Part III System Aspects

Chapter 7 Building Your Own Memory-Resident Areas

7.1 What is a Resident Area? 7–1

7.2 The Steps in Creating a Resident Area 7–1

7.3 How to Build Memory-Resident Areas 7–2
- 7.3.1 Building Position-Independent Resident Areas 7–2
- 7.3.2 Building Absolute Resident Areas 7–3

7.4 Resident Areas with Memory-Resident Overlays 7–4
- 7.4.1 Specifying Memory-Resident Overlays 7–5
- 7.4.2 Building Memory-Resident Overlays 7–6

7.5 Building Your Own Cluster Libraries 7–8
7.5.1 Rule 1: Position Independent or Built for Exact Address 7–8
7.5.2 Rule 2: Use Memory-Resident Overlays 7–8
7.5.3 Rule 3: No Required Parameters on the Stack 7–9
7.5.4 Rule 4: No Trap or Asynchronous Entry 7–9
7.5.5 Rule 5: No Calls to Routines in Another Cluster Library 7–10
7.5.6 Revectoring Cluster Libraries 7–10
7.5.7 Sample Vector Table Code 7–12
7.5.8 GBLXCL and GBLINC Options 7–12

7.6 FORTRAN Virtual Arrays 7–13

7.7 Virtual Program Selectors 7–14
7.7.1 FORTRAN Run-Time Support for Virtual Program Sections 7–17
7.7.1.1 The ALSCT Subroutine 7–18
7.7.1.2 The RLSCT Subroutine 7–19
7.7.2 Building a Program That Uses a Virtual Program Section 7–20

7.8 Advanced Programmed Region Control 7–26
7.8.1 The EXTM$ Feature 7–27

7.9 Fast-Mapping Facility 7–28
7.9.1 Fast-Mapping Code Provided by TKB 7–29
7.9.2 Programming Considerations 7–29
7.9.2.1 Calling Sequence 7–30
7.9.2.2 Returned Data 7–30
7.9.2.3 Programming Examples 7–30

Chapter 8 User-Mode I- and D-Space

8.1 User-Task Data Space 8–1

8.2 I- and D-Space Task Identification 8–1

8.3 Comparison of Conventional Tasks and I- and D-Space Tasks 8–1

8.4 Conventional Task Mapping 8–2

8.5 I- and D-Space Task Mapping 8–3

8.6 Designing an I-and D-Space Task 8–4

8.7 Concurrent Libraries 8–4

Chapter 9 Supervisor-Mode Library

9.1 Mode-Switching Vectors 9–1

9.2 Completion Routines 9–2

9.3 Programming Considerations for the Contents of Supervisor-Mode Libraries 9–2

9.4 Supervisor-Mode Library Mapping 9–2
9.4.1 Supervisor-Mode Library Data 9–3
9.4.2 Supervisor-Mode Libraries with I- and D-Space Tasks 9–3

9.5 Building and Linking to Supervisor-Mode Libraries 9–4
9.5.1 Relevant TKB Options 9–5
9.5.2 Mode-Switching Instruction 9–5
9.5.2.1 Required Memory Layouts for Supported CSM Instructions 9–5
9.5.2.2 The CSM Library Dispatching Process 9–5

9.6 CSM Libraries 9–6
9.6.1 Building a CSM Library 9–6
9.6.2 Linking to a CSM Library 9–8
9.6.3 Example of a CSM Library and Building a Task 9–8
9.6.3.1 Building the Library SUPER 9–16
9.6.3.2 Building TSUP 9–17
9.6.3.3 Running TSUP 9–18
9.6.4 Passing Parameters Using Stack Space 9–18

9.7 Using Supervisor-Mode Libraries as User-Mode Resident Libraries 9–19

9.8 Multiple Supervisor-Mode Libraries 9–19

9.9 Linking Supervisor-Mode Libraries 9–19

9.10 Writing Your Own Vectors and Completion Routines 9–19

9.11 Overlaid Supervisor-Mode Libraries 9–20

9.12 Using ODT to Debug CMS Library 9–20

9.13 Trap Handling with Supervisor Libraries 9–21
9.13.1 Locating Service Routines 9–21
9.13.1.1 FPPA$ AND SCCA$ 9–21
9.13.1.2 SVTK$ and SVDB$ 9–21

9.14 Building to a Supervisor-Mode RMS Library 9–22

9.15 Map Supervisor D-Space 9–23

Part IV Reference Section

Chapter 10 Task Builder Command Line Format

10.1 Running the Task Builder 10–1
10.1.1 Command Line 10–1
10.1.2 Multiline Command 10–3

10.2 Options 10–3

10.3 Multiple Builds in One Run 10–4

10.4 Indirect Command Files 10–4

10.5 Comments in Lines 10–6

10.6 File Specifications 10–6

Chapter 11 Task Builder Switches

11.1 /CC—Concatenated Programs and Subprograms 11–3

11.2 /CO—Build a Common Block Shared Region 11–4

11.3 /DA—Debugging Aid 11–5

11.4 /DL—Default Library 11–6

11.5 /EL—Extend Library 11–7

11.6 /FM—Fast Map 11–8

11.7 /FO—Fast Map Overlay 11–9

11.8 /FP—Floating Point 11–10

11.9 /FU—Full Search 11–11

11.10 /HD—Header 11–12

11.11 /ID—I- and D-Space 11–13

11.12 /LB—Library File 11–14

11.13 /LI—Build a Library Shared Region 11–16

11.14 /MA—Map Contents of File 11–17

11.15 /MP—Overlay Map 11–18

11.16 /MU—Multiuser Program 11–19

11.17 /NM—No Diagnostic Messages 11–20

11.18 /PI—Position Independent 11–21

11.19 /PM—Post-Mortem Dump 11–22

11.20 /RO—Resident Overlay 11–23

11.21 /SB—Slow Build 11–24

11.22 /SG—Segregate Program Sections 11–25

11.23 /SH—Short Map 11–26

11.24 /SP—Spool Map Output 11–32

11.25 /SQ—Sequential 11–33

11.26 /SS—Selective Search 11–34

11.27 /TR—Traceable Program 11–36

11.28 /WI—Wide Listing Format 11–37

11.29 /XT[:n]—Exit on Error 11–38

Chapter 12 Task Builder Options

12.1 ABORT—Abort the Build 12–3

12.2 ABSPAT—Absolute Patch 12–4

12.3 ACTFIL—Number of Active Files 12–5

12.4 ASG—Assign Devices 12–6

12.5 CLSTR—Cluster Libraries 12–6

12.6 CMPRT—Completion Routine 12–9

12.7 COMMON—Access System Common Block 12–10

12.8 DSPPAT—Absolute Patch for D-Space 12–11

12.9 EXTSCT—Extend Program Section 12–12

12.10 EXTTSK—Extend Task Memory 12–13

12.11 FMTBUF—Format Buffer Size 12–14

12.12 GBLDEF—Define a Global Symbol 12–15

12.13 GBLINC—Include Global in .STB File 12–16

12.14 GBLPAT—Global Relative Patch 12–17

12.15 GBLREF—Global Symbol Reference 12–18

12.16 GBLXCL—Exclude Global from .STB File 12–19

12.17 HISEG—Define High Segment 12–20

12.18 LIBR—Access System-Owned Resident Library 12–21

12.19 MAXBUF—Maximum Record Buffer Size 12–23

12.20 ODTV—ODT SST Vector 12–24

12.21 PAR—Partition for Resident Area 12–25

12.22 RESCOM—Access Resident Common Block 12–26

12.23 RESLIB—Access Resident Library 12–27

12.24 RESSUP—Resident Supervisor-Mode Library 12–29

12.25 RNDSEG—Round Segment 12–30

12.26 STACK—Declare Stack Size 12–31

12.27 SUPLIB—Resident Supervisor-Mode Library 12–32

12.28 TASK—Program Name for SYSTAT 12–33

12.29 TSKV—Task SST Vector 12–34

Examples

3–1 Overlay Description of Memory Allocation Map 3–7
4–1 First Page of Map File for Sample Program 4–10
4–2 Excerpts from Map File for Second Build of Sample Program 4–12
4–3 First Page of Map File for Third Build of Sample Program 4–16
7–1 VSECT2.CMD 7–20
7–2 Source Listing for VSECT2.FTN 7–21
7–3 Task Builder Map (Edited) for VSECT2.TSK 7–23
7–4 VSECT.CMD 7–24
7–5 Source Listing for VSECT.FTN 7–25
9–1 Code for SUPER.MAC 9–9
9–2 Memory Allocation Map for Super 9–10
9–3 Completion Routine $CMPCS from SYSLIB.OLB 9–11
9–4 Code for TSUP.MAC 9–14
9–5 Memory Allocation Map for TSUP 9–16
11–1 Memory Allocation (Map) File 11–27
12–1 A Task Using a Virtual Array with the OVR Attribute 12–36

Figures

1–1 Steps in Creating a Program 1–1
1–2 The Task Builder Resolves Global References 1–2
1–3 The Task Builder Constructs the Overlays You Specify 1–4
2–1 You Tell the Task Builder Which Libraries to Include 2–1
2–2 Job Area: Two User Programs 2–3
2–3 Disk and Resident Libraries 2–6
2–4 Clustered Resident Libraries 2–13
3–1 The ODL File Is Your "Blueprint" for Overlays 3–1
3–2 Outlining the Call Structure 3–2
3–3 A Simple Overlay in Memory 3–3
3–4 Outline of First Call Structure for Example 3–8
3–5 Outline of Second Call Structure for Example 3–9
3–6 Separate Paths in an Overlay Structure 3–11
3–7 Resolving Global Symbols 3–13
3–8 Resolving Global Symbols from Disk Libraries 3–14
4–1 Co-Trees Save More Space Than Simple Overlays 4–1
4–2 Putting A and B in the Root 4–2
4–3 A Co-Tree Structure 4–2
4–4 How a Co-Tree Is Loaded During Program Execution 4–4
4–5 Co-trees Save More Space When Pieces Are the Same Size 4–5
4–6 Call Structure for Sample Program 4–8
4–7 First Build Structure for Sample Program 4–9
4–8 Structure for Second Build of Sample Program 4–10
4–9 Sketch of the Structure for Second Build of Sample Program 4–14
4–10 Structure for Third Build of Sample Program 4–15
5–1 The Easiest Way to Use Autoload Indicators 5–1

5–2 The Four-Word Structure of a Vector Autoload 5–2
5–3 An Overlay Structure Without Autoload Vectors 5–3
6–1 The Task Builder Works with Program Sections 6–1
6–2 Allocating Space for Global Program Sections 6–3
6–3 Allocation of Program Sections for IN1, IN2, and IN3 6–4
7–1 Memory-Resident Overlays 7–5
7–2 Using a Null Memory-Resident Overlay 7–9
7–3 Overview of How Inter-Cluster-Library Calls Work 7–11
7–4 VSECT Option Usage 7–16
8–1 Conventional Task Linked to a Region in I- and D-Space System 8–3
8–2 I- and D-space Task Mapping in an I- and D-space System 8–4
9–1 Mapping of a 24K Word Conventional User Task Linking to a 16K Word Supervisor-Mode Library 9–3
9–2 Mapping of a 40K Word I- and D-Space Task Linking to an 8K Word Supervisor-Mode Library 9–4
9–3 Overlay Configuration Allowed for Supervisor-Mode Libraries 9–20
B–1 General Object Module Format B–2
B–2 GSD Record and Entry Format B–4
B–3 Module Name Entry Format B–5
B–4 Control Section Name Entry Format B–6
B–5 Internal Symbol Name Entry Format B–6
B–6 Transfer Address Entry Format B–7
B–7 Global Symbol Name Entry Format B–7
B–8 PSECT Name Entry Format B–8
B–9 Program Version Identification Entry Format B–10
B–10 Mapped Array Declaration Entry Format B–11
B–11 Completion Routine Entry Format B–11
B–12 End-of-GSD Record Format B–12
B–13 Text Information Record Format B–13
B–14 Relocation Directory Record Format B–15
B–15 Internal Relocation Entry Format B–16
B–16 Global Relocation Entry Format B–16
B–17 Internal Displaced Relocation Entry Format B–17
B–18 Global Displaced Relocation Entry Format B–17
B–19 Global Additive Relocation Entry Format B–18
B–20 Global Additive Displaced Relocation Entry Format B–19
B–21 Location Counter Definition B–19
B–22 Location Counter Modification B–20
B–23 Program Limits Entry Format B–20
B–24 PSECT Relocation Entry Format B–21
B–25 PSECT Displaced Relocation Entry Format B–22
B–26 PSECT Additive Relocation Entry Format B–23
B–27 PSECT Additive Displaced Relocation Entry Format B–24
B–28 Complex Relocation Entry Format B–25
B–29 Additive Relocation Entry Format B–26
B–30 General Format of All ISD Records B–27
B–31 General Format of a TKB-Generated Record B–28
B–32 Format of TKB-Generated Start-of-Segment Item (1) B–28
B–33 Format of TKB-Generated Task Identification Item (2) B–29

B–34 Format of an Autoloadable Library Entry Point Item (3) B–30
B–35 Format of a Module Name Item (Type 1) B–31
B–36 Format of a Global Symbol Item (Type 2) B–32
B–37 Format of a PSECT Item (Type 3) B–33
B–38 Format of a Line-Number or PC Correlation Item (Type 4) B–34
B–39 Format of an Internal Symbol Name Item (Type 5) B–35
B–40 Format of a Literal Record Type B–36
B–41 End-of-Module Record Format B–36
C–1 Task Image on Disk C–1
C–2 Label Block Group C–3
C–3 Task Header Fixed Part C–7
C–4 Task Header Variable Part C–8
C–5 Vector Extension Area Format C–10
C–6 Task-Resident Overlay Data Base C–11
C–7 Task-Resident Overlay Database for and I- and D-Space Overlaid Task C–12
C–8 Autoload Vector Entry C–13
C–9 Segment Descriptor C–14
C–10 Sample Tree C–16
C–11 Segment Linkage Directives C–16
C–12 Autoload Vector Entry for I- and D-Space Tasks C–17
C–13 Window Descriptor C–18
C–14 Region Descriptor C–19

Tables

2–1 Disk Libraries Used with RSTS/E 2–4
2–2 Applicable Libraries for the CLSTR Option 2–12
6–1 Program Sections for IN1, IN2, AND IN3 6–4
7–1 Format of Region Descriptor 7–18
7–2 Bit Meanings in the Mask Value 7–26
7–3 Values for the First Fast-Mapping Call Parameter 7–29
8–1 Mapping Comparison Summary 8–2
11–1 Task Builder Switches 11–1
11–2 Input Files for /SS Example 11–34
12–1 Task Builder Options 12–1
B–1 GSD Entry Types B–3
B–2 Types of Entries for Relocation Directory Records B–14
B–3 Defined Operation Codes for the RLD Command Word B–24
C–1 Task and Resident Library Data C–4
C–2 Contents of SRTS/Common Name Block C–6
D–1 Task Builder Reserved Global Symbols D–1
D–2 PSECT Names Reserved by the Task Builder D–3

Preface

Objectives

This manual describes how to use the RSTS/E Task Builder to link your compiled or assembled programs and subprograms into an executable program file to run on RSTS/E.

On RSTS/E systems, your programs must be linked by the Task Builder if they were written in languages for the compilers listed below. Note that this manual is current for the versions shown in parentheses. Information about using the Task Builder may change for subsequent versions.

Compiler (Version)

BASIC-PLUS-2 (V2.6)
FORTRAN-77 (V5.3)
PDP-11 C (V1.0)
COBOL-81 (V3.0)
DIBOL (V6.1)

This manual also applies to the MAC assembler (V5.5) for MACRO programs.

Audience

Although you do not need to be a computer expert to use this manual, you should have a general understanding of computer languages and be familiar with using programs and subprograms.

Document Structure

This manual contains four parts, as indicated by the divider sheets preceding each section, and five appendixes:

Part I	Tells all you need to know to get a program built with the right libraries to run on a RSTS/E system. The two types of libraries (disk libraries and memory-resident libraries) are explained, along with details on how to link them with your program.

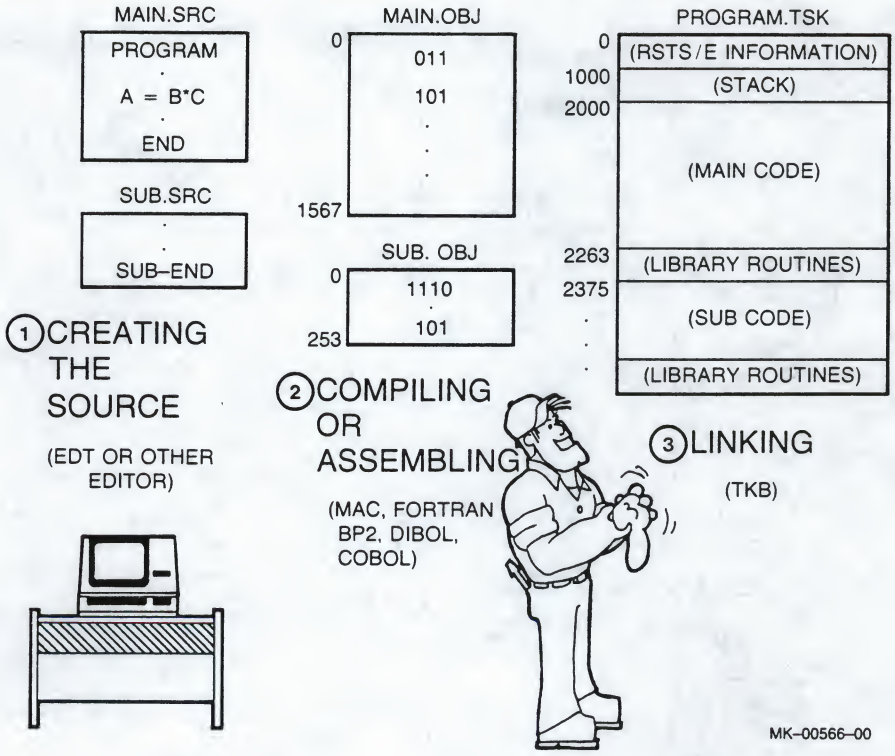
Chapter 1

Introduction

You need to use the Task Builder (TKB) if you write programs on RSTS/E systems in BASIC-PLUS-2, FORTRAN-77, PDP-11 C, COBOL-81, DIBOL, or the MACRO assembly language using the MAC assembler.

The compilers and assemblers associated with these languages translate your programs and subprograms (called source code) into machine instructions (object code). The Task Builder applies the final touches, converting the object code produced by the compilers to code that can be executed by the computer. Figure 1–1 shows the steps involved in creating a program.

Figure 1–1: Steps in Creating a Program



1.1 What the Task Builder Does

The Task Builder handles two basic functions: linking and producing overlays.

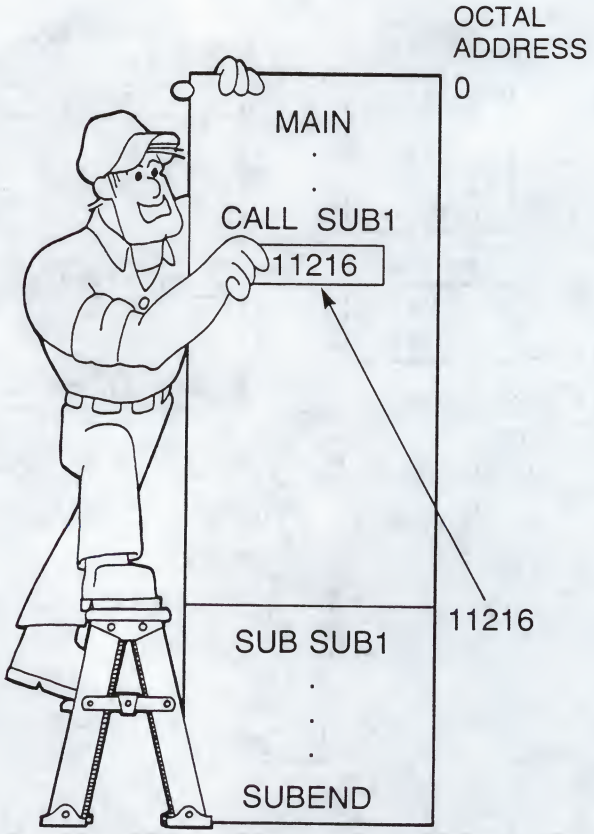
1.1.1 Linking

Linking is necessary because you seldom write programs as one unit. It is easier to work with programs that are written as modules – programs and subprograms—that you can separately design, code, debug, and maintain.

Even if you code your program as one main program, with no separately assembled or compiled subprograms, every compiler translates some source statements into calls to subroutines kept in libraries. For example, all the compilers generate calls to library subroutines to perform I/O or do mathematical calculations. Libraries are provided with the system and with the compilers available with RSTS/E systems.

The Task Builder links these separate modules—your main program, subprograms, and library routines—together in the order you specify, resolving any references that cross module boundaries. For example, Figure 1–2 shows a call to SUB1 from the program MAIN.

Figure 1–2: The Task Builder Resolves Global References




```
RUN $TKB
TKB>MAIN=MAIN, SUB1, LB:F4POTS/LB
TKB>//
```

MK–00567–00

The command to the Task Builder (the line after RUN $TKB) says that these two modules are to be linked together. In addition, any routines necessary from the FORTRAN library are to be linked with these two modules. To simplify, the figure shows only the linking of MAIN and SUB1. Part of the linking process involves generating the proper succession of addresses. As Figure 1–1 showed, the compilers and assemblers generate what are called "relative addresses"; the first address of each module (MAIN and SUB1) is numbered 0 at the compilation stage. When the Task Builder links modules, it changes the addresses of the second and following modules to begin where the addresses of the previous module left off. So, the final addresses for the linked program, as assigned by the Task Builder, range upward from 0 in succession.

The second aspect of linking is resolving references to what are called "global symbols." At compile time, for example, MAIN's reference to SUB1 cannot be resolved. SUB1 is flagged as a global reference (somewhere in the "world outside of MAIN") when MAIN is compiled. Likewise, when SUB1 is compiled, it is again flagged as a global symbol; it will serve as an entry point from the "outside world."

The Task Builder, as shown in Figure 1–2, keeps track of the addresses assigned to global symbols and substitutes the address for the entry point of SUB1 into the call in MAIN. Then, when the program is run, and the call is executed, control transfers to address 11216, the entry point for SUB1.

1.1.2 Overlays

The second necessary service that the Task Builder provides is a means to construct overlays. The amount of memory from which programs can be executed is limited on PDP-11 computers to 32,000 words. On RSTS/E systems, for reasons described in Chapter 2, there are further limitations. If your program is too large to fit in the space available, you must specify how you want it overlaid—such that sections of code and data can be called into memory at different times (the new sections "overlaying" the old).

Figure 1–3 shows the concept behind overlays. The Task Builder links both the modules SUB1 and SUB2 to start at address 15,726. The Task Builder then inserts code into MAIN such that, when MAIN's call to SUB2 is executed, SUB2 will replace SUB1, called and executed previously. SUB1 does not have to be the same length as SUB2, but both will be linked to start at the same address.

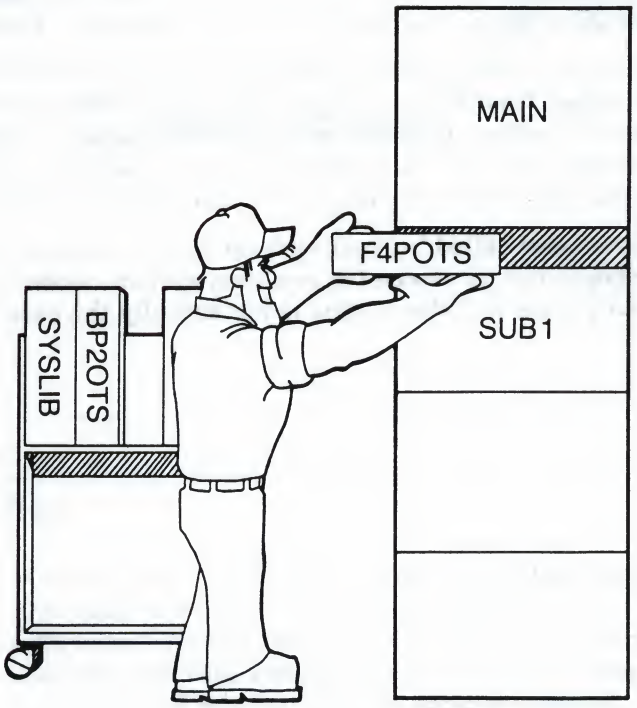
Figure 1-3 also shows something called the "high segment" in high address space. This code is the main reason your program does not have the full 32,000 words available on PDP-11 systems. For further information, see Chapter 2.

Chapter 2

Building Programs

This chapter tells how to build nonoverlaid programs. How large can a program be before it must be overlaid? The answer depends on the language you used to write your program; Section 2.1 discusses some specifics. The library routines built into your executable program also affect its size (Figure 2–1). Section 2.2 names and describes the disk libraries currently provided by Digital for the various languages. Section 2.3 discusses the Task Builder command line in general, and Section 2.4 gives specific examples for building programs written in each of the various languages.

Figure 2–1: You Tell the Task Builder Which Libraries to Include




```
RUN $TKB
TKB>MAIN,SUB1,LB:F4POTS/LB
TKB>//
```

MK–00569–00

2.2.1 Disk Libraries

The libraries listed in Table 2–1 are currently shipped with RSTS/E and its associated languages. Note that the table is current for the versions of the software mentioned in the Preface. As new versions of languages are released, library names and contents may change. In addition, other products available with RSTS/E can have associated libraries, and your own installation may have generated its own libraries.

One way to find out what libraries are available is to get a directory of the system library device (LB:) with a wildcard file name and a file type of .OLB. (OLB stands for object library.) For example:

```
DIR LB:*.OLB

Name .Typ    Size   Prot       DR3:[1,1]
SYSLIB.OLB   220    < 40>
RMSLIB.OLB   300    < 40>
BP2OTS.OLB   225    < 40>
COBLIB.OLB   178    < 40>
```

Table 2–1 describes some of the libraries in this account that your program may use.

Table 2–1: Disk Libraries Used with RSTS/E

Disk Library Name	Description
SYSLIB.OLB	The system library. Contains many routines used by programs written in MACRO (for the MAC assembler) and the higher-level languages. The Task Builder always searches this library to resolve undefined symbols. You do not need to specify it in a Task Builder command line.
RMSLIB.OLB	Contains routines needed if you use RMS (Record Management Services) on RSTS/E systems.
RMSDAP.OLB	Contains routines needed for network record access through RMS on RSTS/E systems.
BP2OTS.OLB	Contains routines needed to run your BASIC-PLUS-2 program under the RSX run-time system.
DBLLIB.OLB	Contains routines needed to run your DIBOL program if it uses the DIBOL Management System (DMS) for I/O. Note that you must also declare a resident library (DBLRES) if you use this disk library. See Section 2.3.4 for information on how to specify resident libraries.
DBRLIB.OLB	Contains routines needed to run your DIBOL program if you use the Record Management System (RMS) for I/O. Note that you must also declare a resident library (DBRRES) if you use this disk library. See Section 2.3.4 for information on how to specify resident libraries.
COBLIB.OLB	Contains routines needed to run your PDP-11 COBOL program. If you use this library rather than COBOVR.OLB, your program will take more memory but will run faster.

(continued on next page)

Table 2–1 (Cont.): Disk Libraries Used with RSTS/E

Disk Library Name	Description
COBOVR.OLB	Contains routines needed to run your PDP-11 COBOL program if it is overlaid. You use this library if you use the PDP-11 COBOL segmentation facility. However, if you use this library rather than COBLIB.OLB, your program will run slower, as the routines are called in as needed and overlay each other.
C81CIS.OLB	Contains routines needed to run your COBOL-81 program if the program was compiled with the /CIS switch. This is the normal default if your computer has the Commercial Instruction Set (CIS) option.
C81LIB.OLB	Contains routines needed to run your COBOL-81 program if the program was compiled with the /-CIS switch. This is the normal default if your computer does not have the Commercial Instruction Set (CIS) option.
FDVDBG.OLB	Contains routines needed if you use the FMS form driver with debug mode support.
FDVLIB.OLB	Contains routines needed if you use the FMS form driver without debug mode support.
F4POTS.OLB	Contains routines needed to run your FORTRAN-77 program.
F4PRMS.OLB	Contains routines for FORTRAN-77 programs using RMS (Record Management Services) for I/O.

2.2.2 Resident Libraries

In addition to disk libraries, you may also have to work with resident libraries on RSTS/E systems. "Resident" means residing in computer memory. The system manager defines libraries as resident so that they can be shared by more than one user. Instead of building routines into your program (as is done with disk libraries), you use a copy of the library. The copy is resident in memory as long as you or someone else is using it.

The Task Builder links your program to appropriate routines in the resident library by a technique called "mapping." Mapping is the process of accessing different logical areas of memory. With the mapping technique, many programs can use routines from the same space in computer memory. The system manager usually defines a library to be resident when it is heavily used. In such cases, less overall computer memory is taken by a resident library than by having each program include its own copy of routines from the library.

Figure 2–3 shows the difference between disk and resident libraries. For disk libraries, the Task Builder takes a copy of each routine that you reference in your program and builds it into your program. Note that a copy of RTNA has been built into both PROG1 and PROG2 in this figure. However, both programs can reference a resident library from the same area of physical memory.

The above sequence produces the same result as the single-line command:

```
TKB IMG1,IMG1,IMG1=JOHN:FILE1,SY:FILE2,FILE3,LB:RMSLIB/LB,SY:MYLIB/LB
```

You must specify the output file specifications and the equal sign on the first line. You can begin or continue input file specifications on subsequent lines.

2.3.3 Options

You may need to specify options to build a particular program. An option modifies the action taking place during the build. To include options, you must use the multiline format as shown below. When you type a line consisting of a single slash (/), the Task Builder assumes that the last input file has been entered and prompts for options by displaying "ENTER OPTIONS:" and another "TKB>" prompt.

```
RUN $TKB
TKB>command
TKB>continued-command
TKB>/
ENTER OPTIONS:
TKB>option=value:value
TKB>//
```

The format for options is shown here because some languages require certain options for a Task Build. If your language manual set includes a user's guide, you will probably find helpful pointers about necessary or particularly useful options for your language. Table 12–1 in the Reference Section of this manual (Part IV) gives an overview of all the options available for the Task Builder. The options are then described in detail in the remainder of Chapter 12.

The options you will probably find most useful regardless of source language are RESLIB and LIBR. You need to use these options if you need to link to one or more resident libraries. Since resident libraries are commonly used, these options are discussed in the following section. Some examples of these and other options are shown in Section 2.4.

2.3.4 The LIBR and RESLIB Options

You can link to a maximum of seven user mode resident libraries using the Task Builder on RSTS/E systems. Supervisor mode resident libraries are explained in Chapter 9. With either the LIBR or RESLIB option, you specify that you want to link your program to one resident library. The choice between LIBR or RESLIB depends on whether the library is "system-owned" or "user-owned."

The LIBR option declares that your program intends to access a "system-owned" resident library. "System-owned" simply means that the file containing the library is located in the library account (LB:). This can be any account on any disk, as assigned by the system manager.

"User-owned" means that the library can be on some disk or account other than LB:. With the RESLIB option, you specify the disk containing the resident library files.

The formats for the options are:

LIBR=name:access-code[:apr]

RESLIB=file-specification/access-code[:apr]

Note that with the LIBR option, you name only the resident library. The Task Builder looks for the appropriate files (name.STB and name.TSK) on the system library disk (LB:) when it is building the code necessary to load the resident library. With the RESLIB option, you specify a complete file specification. This names the device, account, and file name of the executable file to be loaded. You do not specify the file type. The Task Builder uses the executable file and the symbol table file for the library, and requires that they have file types of .TSK and .STB.

The access-code is either RW (read/write) or RO (read-only), indicating how your program intends to access the library. (It will be RO for Digital-provided resident libraries such as RMSRES.)

The Active Page Register (APR) parameter is an integer in the range of 1 to 7 that specifies the first APR reserved for the library. If you leave this parameter off, the Task Builder assigns the highest APR it can to the resident library.

It is not really necessary to understand Active Page Registers to understand or use the APR modifier. Think of your 32K word user job area as divided into eight parts of 4K words each, numbered from 0 through 7. Your program occupies one or more of the lowest-numbered segments.

You can "map" resident libraries into the area between the top of the program and the highest address of virtual memory. The map must begin on a 4K-word boundary. For example, suppose your program takes 6K words and the run-time system takes 4K words of memory. You can map up to 20K words of resident library into your job, beginning with APR 2.

NOTE

With the use of advanced programming techniques, it is possible to use resident libraries with certain run-time systems other than RSX. However, Digital supports the use of resident libraries only under the RSX run-time system.

2.3.5 The CLSTR Option

You can use the CLSTR option if you need to use more than one resident library. CLSTR lets multiple resident libraries share the same virtual address space in your program. However, not all resident libraries available with RSTS/E can take advantage of this feature. Table 2-2 lists those libraries to which CLSTR can apply.

Chapter 3

The Basic Concepts

If your program is too large to fit in the space available, you must specify an overlay structure for it. The easiest way to find out if your program is too large is to try to build it, using the steps outlined in Chapter 2. If you get the following error message, your program is too large:

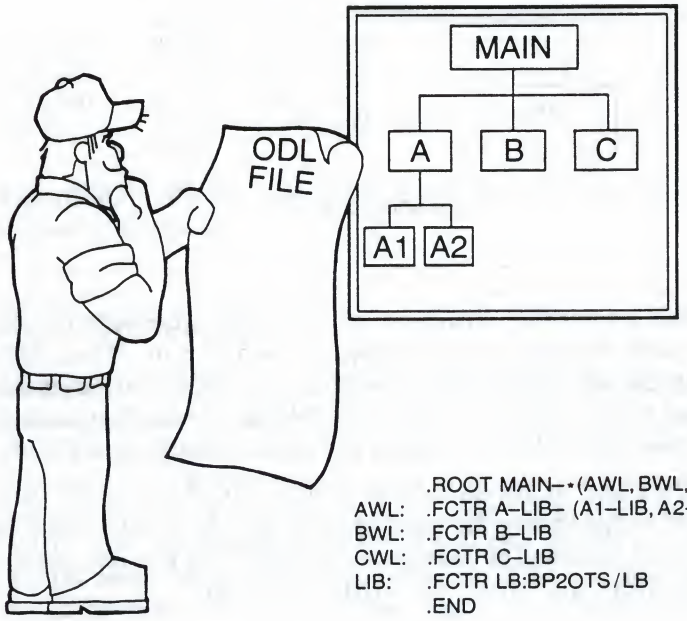
```
?Task has illegal memory limits
```

Languages that can dynamically allocate memory (such as BASIC-PLUS-2) may not give this error at task build. Rather, they may produce another message at run time, such as:

```
?Maximum memory exceeded
```

This chapter tells how to specify an overlay structure to eliminate this problem. You design an overlay structure, such as the diagram in Figure 3–1, and describe the structure to the Task Builder using an "ODL file"; a file written in the Overlay Description Language.

Figure 3–1: The ODL File Is Your "Blueprint" for Overlays




```
      .ROOT MAIN-*(AWL,BWL,CWL)
AWL:  .FCTR A-LIB- (A1-LIB,A2-LIB)
BWL:  .FCTR B-LIB
CWL:  .FCTR C-LIB
LIB:  .FCTR LB:BP2OTS/LB
      .END
```

MK-00573-00

The syntax of the .ROOT and .FCTR commands defines the overlay structure. The first item following the .ROOT command indicates the root item, to be assigned the lowest virtual addresses:

```
.ROOT  MAINWL-*(SUB1WL,SUB2WL)
```

The root item in this example is MAINWL. This item—named to denote "MAIN With Library"—is defined in the following .FCTR command:

```
MAINWL:    .FCTR  MAIN-LIBR
```

For the moment, however, consider the .ROOT command. The following symbols define the structure of the overlay:

-	Separates pieces to be concatenated in memory
,	Separates pieces to be overlaid in memory
( )	Groups pieces to be overlaid

Thus, the hyphen in the .ROOT command indicates that MAINWL is to be concatenated with the structure (SUB1WL,SUB2WL). The parentheses indicate grouping; they enclose items that are to overlay each other. The structure inside—SUB1WL,SUB2WL—indicates SUB1WL and SUB2WL are to occupy the same space, or overlay each other as necessary.

In other words, a comma separating two or more items within parentheses indicates that they are to overlay each other. A dash between two items indicates they are to be concatenated, with the item on the left assigned the lowest addresses.

The asterisk (*) symbol shown in the example is an autoload indicator. It does not affect the overlay structure, although it is very important. It tells the Task Builder to generate what are called autoload vectors to ensure that overlay pieces can be loaded properly when the program is executed.

The use of asterisks is discussed in detail in Chapter 5; you can save a little space in your program if you use them carefully. However, the simplest rule, and one that always ensures proper loading for overlay structures described in this chapter, is to put an asterisk before the outermost left parenthesis in your ODL file.

3.2.2 The .FCTR Command

Consider the example under discussion again:

```
          .ROOT  MAINWL-*(SUB1WL,SUB2WL)
MAINWL:   .FCTR  MAIN-LIBR
SUB1WL:   .FCTR  SUB1-LIBR
SUB2WL:   .FCTR  SUB2-LIBR
LIBR:     .FCTR  LB:BP2OTS/LB
          .END
```

MAINWL, SUB1WL, and SUB2WL are all defined as factors in the lines following the first line. The term "factor" is used in the sense of "ingredient." That is, .FCTR commands are used to further define elements used in a .ROOT command or a preceding .FCTR command.

Note that the names used in the .ROOT command are defined in each .FCTR command by the first field: the name terminated by a colon. Likewise, the name LIBR, used in several of the .FCTR commands, is defined in the last .FCTR command. In general, factor names can consist of 1-6 characters from the set A-Z, 0-9, and the dollar sign ($).

The .FCTR command also specifies an overlay structure; the same items and operators used in a .ROOT command can also be used in a .FCTR command. In the example, the first three .FCTR commands consist of two items separated by a hyphen. Again, the hyphen separating two items means that the first item is assigned the lowest addresses, and the second item is to be concatenated following the first.

In the example shown at the start of Section 3.2, however, the concatenated item is LIBR, defined by a later .FCTR command as the BP2OTS library. When an item in a hyphenated series is a file with the /LB switch, it means that the first item's unresolved references are to be resolved from routines within that disk library. In other words, the entire library is not concatenated. Only those routines referenced are actually concatenated and added to the executable file. The items MAIN, SUB1, and SUB2 are the compiled or assembled object files. As with a simple build, the default file type for such files is .OBJ. The default file type for a file with the /LB switch is .OLB.

Note that a .FCTR command can contain an item defined in another .FCTR command. In general, .FCTR commands can be "nested" in this fashion up to 16 levels.

3.2.3 The .END Command

The .END command ends the ODL file; every ODL file must have one .ROOT command and end with .END.

3.2.4 Flexibility of the Overlay Description Language

From the preceding discussion, you probably have observed that there are many ways to construct ODL files using the three basic commands and their operators. For example, the following ODL file has the same effect as the example in Section 3.2:

```
.ROOT MAIN-LB:BP2OTS/LB-*(SUB1-LB:BP2OTS/LB,SUB2-LB:BP2OTS/LB)
.END
```

The ODL file above has no .FCTR statements. The following file also produces the same structure as the example at the beginning of Section 3.2:

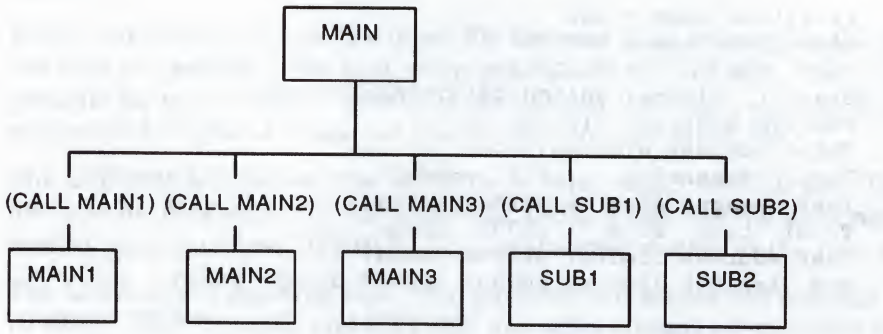
```
        .ROOT MAIN-LIBR-*(SUB1-LIBR,SUB2-LIBR)

LIBR:  .FCTR LB:BP2OTS/LB
       .END
```

3.5.1 Considering Space: Two Possibilities for Example

Suppose that examining the program in our example reveals two possibilities for dividing the program so that the pieces will fit.

In the first case, you divide MAIN into five parts by inserting calls in MAIN in the source code. Now you have a "root" segment, MAIN, with five branches, MAIN1, MAIN2, MAIN3, SUB1, and SUB2. The call structure is outlined in Figure 3–4.

Figure 3–4: Outline of First Call Structure for Example



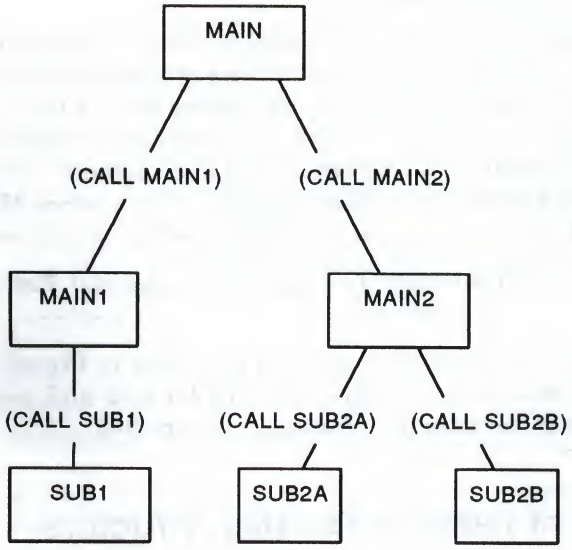
MK-00576-00

Note again the logical independence of the call structure for the items to be overlaid. Defining the overlay structure based on the call structure is one way to ensure the logical independence of the items in the overlay structure. In this case, MAIN1, MAIN2, MAIN3, SUB1, and SUB2 could not call each other or refer to data in each other. These items overlay each other and will not reside in memory at the same time. The ODL file for such a structure could look as follows:

```
        .ROOT MAINWL-*(MAIN1L,MAIN2L,MAIN3L,SUB1L,SUB2L)
MAINWL: .FCTR MAIN-LIBR
MAIN1L: .FCTR MAIN1-LIBR
MAIN2L: .FCTR MAIN2-LIBR
MAIN3L: .FCTR MAIN3-LIBR
SUB1L:  .FCTR SUB1-LIBR
SUB2L:  .FCTR SUB2-LIBR
LIBR:   .FCTR LB:BP2OTS/LB
        .END
```

In the second case, you divide MAIN into two pieces and divide SUB2 into two pieces called SUB2A and SUB2B. The outline for the call structure is shown in Figure 3–5.

Figure 3–5: Outline of Second Call Structure for Example



MK-00577-00

The ODL file for such a structure could look as shown below. Note the nested parentheses used to group the pieces that overlay each other. In general, parentheses can be nested to 16 levels.

```
        .ROOT   MAINWL-*(MAIN1L-SUB1L,MAIN2L-(SUB2AL,SUB2BL))
MAINWL: .FCTR   MAIN-LIBR
MAIN1L: .FCTR   MAIN1-LIBR
SUB1L:  .FCTR   SUB1-LIBR
MAIN2L: .FCTR   MAIN2-LIBR
SUB2AL: .FCTR   SUB2A-LIBR
SUB2BL: .FCTR   SUB2B-LIBR
LIBR:   .FCTR   LB:BP2OTS/LB
        .END
```

Now suppose you build the program successfully in both of the above cases. The problem with space is resolved with either the structure shown in Figure 3–4 or in Figure 3–5. You would choose the structure that requires the least time to execute, as described in the following section.

3.5.2 Considering Time: Reducing Disk Access

When you ask for overlays, the Task Builder inserts code into your program to load the overlays properly. For the example in Figure 3–4, the Task Builder inserts code into MAIN to load MAIN1, MAIN2, MAIN3, SUB1, and SUB2 from disk into memory when they are called. (MAIN itself is loaded by the run-time system when the program is first run.)

Thus, when MAIN calls MAIN1, the code inserted by the Task Builder is executed to load MAIN1 from disk into memory for execution. When MAIN calls MAIN2, this code is again executed to load MAIN2 from disk into memory, and so forth. These disk accesses take time. You want to design your overlays to reduce the number of disk accesses.

3.7.2 Undefined, Multiply Defined, and Ambiguously Defined Global Symbols

The Task Builder resolves references to global symbols at build time. In general, you can define two global symbols with the same name if they are on separate paths and are not referenced from a piece that is common to both paths.

If you define a global symbol on one path but refer to it on another path, the symbol is diagnosed as undefined where it is referenced.

If you define two global symbols with the same name on the same path, the symbol is multiply defined.

If you define two global symbols with the same name on different paths, but the symbol is referenced from a piece that is common to both, the symbol is ambiguously defined.

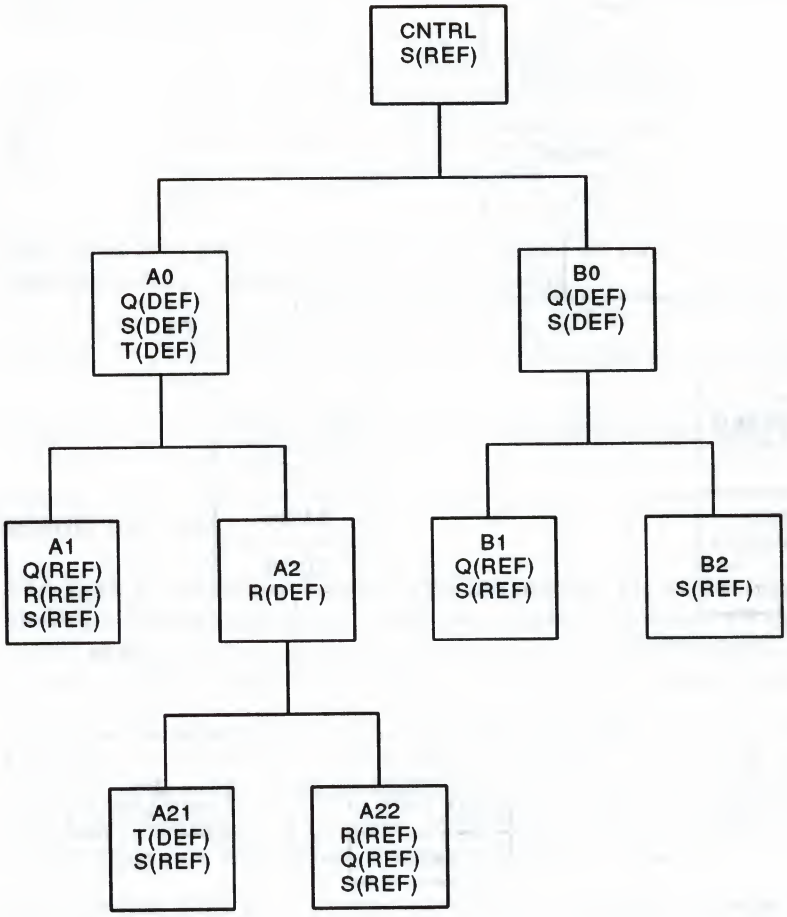
Examine the overlay structure in Figure 3–7. The global symbol Q is defined in A0 and B0. The references to Q in A22 and A1 are resolved by the definition in A0. The reference to Q in B1 is resolved by the definition in B0. The two definitions of Q are distinct in all respects; the definitions and references occupy separate paths.

The global symbol R is defined in A2. The reference to R in A22 is resolved by the definition in A2 because there is a path to the reference from the definition (CNTRL-A0-A2-A22). The reference to R in A1, however, is undefined because there is no definition for R on a path through A1.

The global symbol S is defined in both A0 and B0. References to S from A1, A21, or A22 are resolved by the definition in A0. References to S in B1 and B2 are resolved by the definition in B0. However, the reference to S in CNTRL cannot be resolved, because there are two different definitions of S on separate paths through CNTRL. The global symbol S is ambiguously defined.

The global symbol T is defined in both A21 and A0. Since there is a single path through the two definitions (CNTRL-A0-A2-A21), the global symbol T is multiply defined.

Figure 3–7: Resolving Global Symbols



MK-00579-00

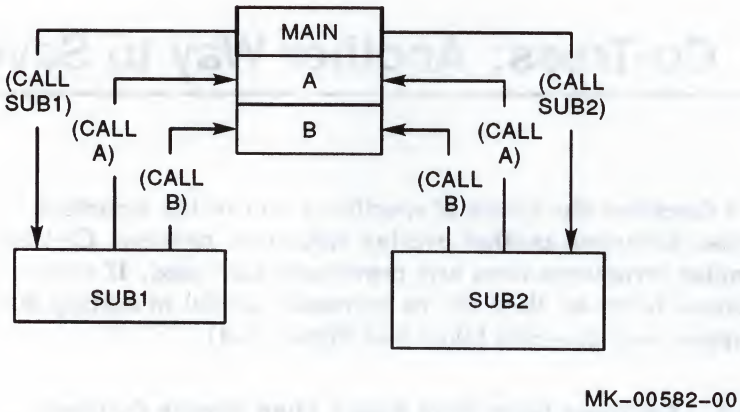
3.7.3 How Routines Are Inserted from Libraries

In all the examples so far, we have shown a library concatenated (using the dash in the ODL file) at the end of the root and every segment in the overlay structure. Unless you know which library routines are used by each piece of your program, this is the best way to ensure that library routines are properly inserted from the desired disk libraries. The Task Builder then ensures that routines referred to by more than one piece are accessible to all pieces. For example, consider the following ODL file for the overlay structure shown in Figure 3–8:

```
        .ROOT   ROOTL-*(AL,BL-(B1L,B2L))
ROOTL:  .FCTR   ROOT-LIBR
AL:     .FCTR   A-A1-LIBR
BL:     .FCTR   B-LIBR
B1L:    .FCTR   B1-LIBR
B2L:    .FCTR   B2-LIBR
LIBR:   .FCTR   LB:F4POTS/LB
        .END
```

of the root, so that they are always accessible from any branch of the tree (see Figure 4–2).

Figure 4–2: Putting A and B in the Root

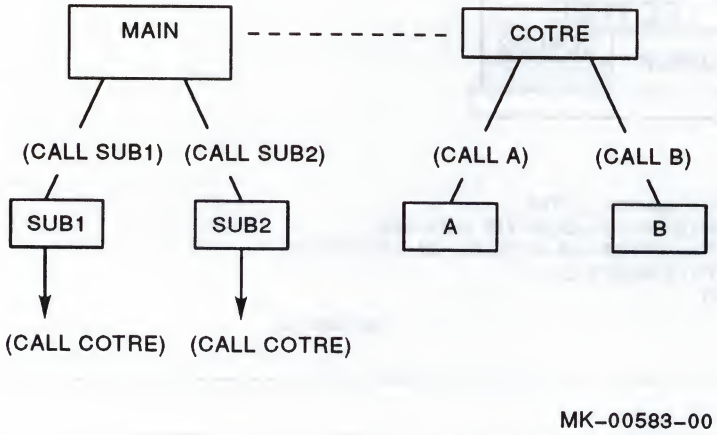


The ODL file for such a structure could appear as follows:

```
       .ROOT MAIN-A-B-LIBR-*(SUB1-LIBR,SUB2-LIBR)
LIBR:  .FCTR LB:BP2OTS/LB
       .END
```

Since A and B never call each other, however, they do not need to reside in memory at the same time. To save space, you could define A and B as part of a co-tree, such that they overlay each other. Like the main tree, co-trees must have a root. In this case, the root is called "COTRE" (see Figure 4–3).

Figure 4–3: A Co-Tree Structure



The ODL file for such a structure could be:

```
        .NAME   COTRE
        .ROOT   MANTRE,COTREE
MANTRE: .FCTR   MAIN-LIBR-*(SUB1-LIBR,SUB2-LIBR)
COTREE: .FCTR   *COTRE-LIBR-*(A-LIBR,B-LIBR)
LIBR:   .FCTR   LB:BP2OTS/LB
        .END
```

To separate co-trees, use the comma—not enclosed in parentheses – as between MANTRE and COTREE in the .ROOT statement above. (When the comma is used within parentheses, it separates pieces to be overlaid.) Note also that you put an autoload indicator (*) before the co-tree root and before the outermost left parenthesis in the co-tree overlay description.

To get an idea of how co-trees are loaded during execution, see Figure 4–4. This figure assumes that the call sequence is: MAIN calls SUB1, which calls COTRE twice: once to execute A and once to execute B. MAIN then calls SUB2, which calls COTRE to call A and B again.

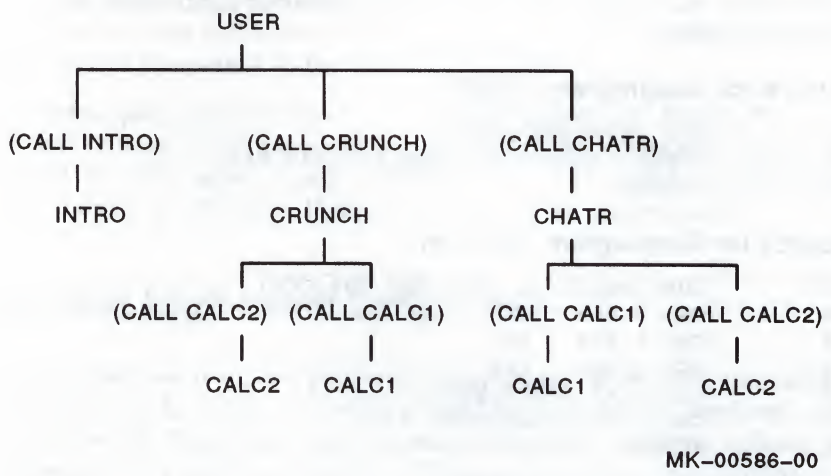
The run-time loader loads MAIN. The remaining pieces are loaded by code inserted in MAIN by the Task Builder. Once called (at time 3 in Figure 4–4), COTRE is resident in memory for the rest of the run. Note that it begins at the place where the longest part of the main tree ends (after SUB2).

As shown in Figure 4–4, storage is not shared between trees. Any piece in a tree can call or refer to data in another tree without displacing pieces from the calling tree. However, calls back and forth between trees (called cross-tree calls) can cause problems. For example, suppose that at time 4 in Figure 4–4 the subprogram A had called SUB2. SUB2 would be loaded, displacing SUB1 from the main tree. In the normal course of events, SUB2 would return control to A, which would return control to an address generated for SUB1 at build time. SUB1 is no longer in memory, however. Control would be passed to some location in SUB2, which has displaced SUB1 in memory.

To keep this from happening inadvertently, the Task Builder restricts its search through the structure for references to the default library if you specify co-trees. The Task Builder makes one pass through the entire structure trying to resolve global symbols from the pieces you have specified in the ODL file. If there are unresolved symbols after this first pass, the Task Builder makes another pass, attempting to resolve undefined symbols from the default system library. If you have specified co-trees, the Task Builder restricts its search during this second pass.

For example, if the Task Builder resolves a symbol in one tree by inserting a module from the default system library, it does not search through co-trees to see if there are other unresolved references to this module. It restricts its search to the current tree and the root of the main tree. This procedure eliminates cross-tree calls like that described above; necessary code is not inadvertently displaced.

Figure 4–6: Call Structure for Sample Program



4.4.3 Compiling the Sample Program and Subprograms

The general steps for compiling a BASIC-PLUS-2 program are:

```
RUN $BASIC2
BASIC2  <------------  (the prompt from BASIC-PLUS-2)
OLD source-file
BASIC2  <------------
COMPILE  /OBJ
```

For example, to compile a source file named USER.B2S, type:

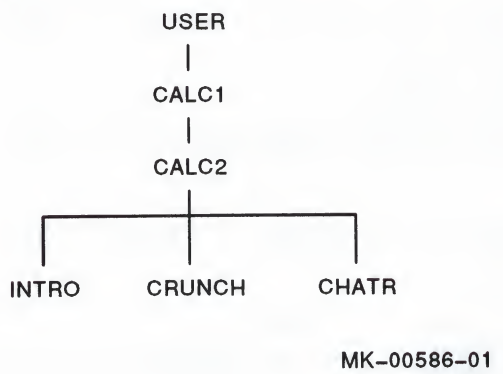
```
RUN $BASIC2
BASIC2
OLD USER
BASIC2
COMPILE /OBJ
```

These commands compile the file USER.B2S, creating the file USER.OBJ. (.B2S and .OBJ are the default file types assumed by BASIC-PLUS-2.)

4.4.4 First Build for Sample Program: Putting Subprograms in the Root

After creating the object files, the next step is task building. For the first build, without co-trees, we put CALC1 and CALC2 in the root (Figure 4–7).

Figure 4–7: First Build Structure for Sample Program



The ODL file for such a structure could be:

```
          .ROOT USER-CALC1-CALC2-LIBR-*(INTWL,CHATWL,CRUNWL)
INTWL:    .FCTR INTRO-LIBR
CRUNWL:   .FCTR CRUNCH-LIBR
CHATWL:   .FCTR CHATR-LIBR
LIBR:     .FCTR LB:BP2OTS/LB
          .END
```

Calling the above file OVER1.ODL, the build process is:

```
RUN $TKB
TKB>TRY1,TRY1=OVER1/MP
ENTER OPTIONS:
TKB>UNITS=12
TKB>ASG=SY:5:6:7:8:9:10:11:12
TKB>EXTTSK=512
TKB>//
```

The build proceeds without error. Examining the map file, TRY1.MAP, you see the first page shown in Example 4–1.

The significant information is highlighted in Example 4–1. The TASK IMAGE SIZE is 6528 words, or 13056 bytes. This means that the total amount of virtual address space that the program will take is 13056 bytes. Further down, the size is itemized by segment. Segment USER (constructed from the files USER.OBJ, CALC1.OBJ, and CALC2.OBJ, plus library routines from BP2OTS.OLB) requires 11348 bytes; INTRO, 1696 bytes; CHATR, 380 bytes; and CRUNCH, 196 bytes. In subsequent builds, you will see the structure (shown by the way the segment names are indented) and the sizes change.

In addition to specifying the main program and subprograms, you also want to build the library routines $ICINI, $ICWRT, $ECONV, $ICFNS, and $STFN1 into the root. Do this by using the /LB switch, followed by specific routine names separated by colons. For example:

```
          .NAME NULL
          .ROOT USERWL,COTRWL
COTRWL:   .FCTR NULL-*(CALC1-LIBR,CALC2-LIBR)
USERWL:   .FCTR USER-LIB-*(INTWL,CRUNWL)
INTWL:    .FCTR INTRO-LIBR
CRUNWL:   .FCTR CRUNCH-CHATR-LIBR
LIB:      .FCTR LIB1-LIBR
LIB1:     .FCTR LB:BP2OTS/LB:$ICINI:$ICWRT:$ECONV:$ICFNS:$STFN1
LIBR:     .FCTR LB:BP2OTS/LB
          .END
```

In general, you can specify up to eight routines as modifiers to one LB switch.

Calling this file OVER3.ODL, the build process is:

```
RUN $TKB
TKB>TRY3,TRY3=OVER3/MP
ENTER OPTIONS:
TKB>UNITS=12
TKB>ASG=SY:5:6:7:8:9:10:11:12
TKB>EXTTSK=512
TKB>//
```

The build proceeds without error. The first page of the map file TRY3.MAP is shown in Example 4–3. As shown, the total amount of virtual address space taken by the program is 6400 words, smaller than the first build without co-trees, which took 6528 words. And, as can be determined by typing RUN TRY3.TSK, the program runs.

Example 4–3: First Page of Map File for Third Build of Sample Program

```
TRY3.TSK    Memory allocation map  TKB 08.006      Page 1
                      15-MAY-90    15:24

Partition name : GEN
Identification : 000708
Task  UIC      : [1,196]
Stack    limits: 001000 001777 001000 00512.
PRG xfr address: 022252
Total address windows: 1.
Task extension      : 512. words
Task  image  size   : 6400. words
Total  task  size   : 6912. words
Task address limits: 000000 030773
R-W disk blk limits: 000002 000037 000036 00030.

TRY3.TSK Overlay description:

Base    Top          Length
----    ---          ------
000000  025253  025254  10924.        USER
025254  030513  003240  01696.             INTRO
025254  026603  001330  00728.             CRUNCH
030514  030513  000000  00000.        NULL
030514  030773  000260  00176.             CALC1
030514  030763  000250  00168.             CALC2
```

4.4.7 Further Tips

The example program discussed in the previous sections is a simple one. For a complex program having many overlays in many co-trees, some of the steps described above are harder to follow; it may be very tedious to write down all the routine names in the diagnostic error messages resulting from a "first-try" co-tree build.

One way to get around this problem is to eliminate all library references in your preliminary build. The routines and symbols will then show up on the map file listing as "undefined symbols," and you can work from there.

4.4.8 Using Co-Tree Techniques with the Default Library

You can use the techniques discussed in this chapter for default library routines. However, you must be even more careful than you were with the language libraries. Routines in the default library were coded in the MACRO assembly language using expert manipulation of program sections (see Chapter 6). For example, a routine may use a data section that can be overlaid in low virtual address space while instruction sections are built into separate paths of the tree. Unless you are a MACRO programmer, aware of these program sections and capable of dealing with them, you should not try overlaying routines from the default library.

If you do want to try it, you will find the /MA and /FU switches useful. These switches are described in detail in Chapter 11.

Briefly, the /MA switch appended to the map file specification calls for more detail in the listing on routines built into the program from the default library.

Appending the /FU switch to the executable program file (default extension .TSK) tells the Task Builder to make a "full search" during its second pass to resolve undefined symbols from the default library. Suppose, for example, that the Task Builder builds a definition from the default library into one path on the main tree to resolve an undefined symbol. If this symbol is referred to from a path on a co-tree and the /FU switch has been used, the Task Builder resolves the reference in the co-tree with the definition in the main tree.

Note that if the symbol were referred to in more than one path on more than one tree, the Task Builder proceeds as it does when it searches through library references specified with a general-purpose /LB switch. That is, it builds the piece into all paths on all trees and flags the symbols as multiply or ambiguously defined. You can specify where you want individual program sections by using the Task Builder's .PSECT command (Chapter 6).

Again, you should not try to overlay pieces from the default library unless you are experienced in MACRO and can determine that cross-tree calls will not inadvertently displace portions of the overlay structure.

this address in the transfer-of-control instruction. For example, consider the simple build:

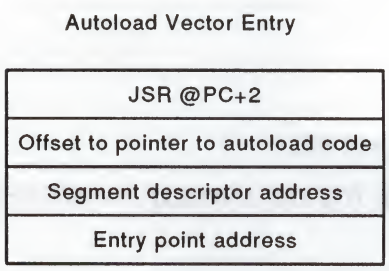
```
RUN $TKB
TKB>OBJ=MAIN,SUB1,LB:F4POTS/LB
TKB>//
```

As described in Chapter 3, the Task Builder concatenates MAIN and SUB1 and resolves undefined references by concatenating modules from the library. When MAIN calls SUB1, the Task Builder resolves the reference by substituting the address it has calculated for the entry point for SUB1.

With overlays, the Task Builder does not assume that such direct substitutions will work. When a call is made to a piece further away from the root, there is no guarantee that the piece referenced will be in memory when the call is executed. So, you must tell the Task Builder to generate autoload vectors for global symbols outside the root that are referenced in transfer-of-control statements by a piece closer to the root. And, where such a reference is made to a global symbol, the Task Builder will then substitute the address of the autoload vector instead of the direct address reference.

The generated autoload vectors are stored in every piece of your program that calls another piece further away from the root. The general form of an autoload vector is the four-word structure shown in Figure 5–2.

Figure 5–2: The Four-Word Structure of a Vector Autoload



MK–01055–00

The JSR instruction passes control to the autoload processor, $AUTO. These two words are followed by the address of the descriptor for the segment to be loaded as well as the direct address calculated for the entry point of the piece to be loaded, if necessary.

Thus, when your program executes a call to a piece of your program further away from the root, control transfers to the autoload vector address and on to the autoload routine (inserted as a part of your program by the Task Builder). The autoload routine checks to see if the piece referred to is already in memory. If so, control is transferred to the location where the called routine resides, that is, to the entry point specified in the last word of the autoload vector.

If the desired piece is not currently in memory, the autoload routine loads the piece from disk (using information from the segment descriptor pointed to by the third word of the autoload vector). Once the appropriate piece is loaded, control is transferred to the entry point specified in the last word of the autoload vector.

5.2 Where are Autoload Vectors Really Needed?

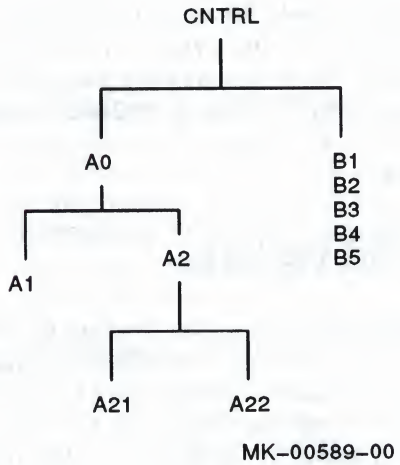
You can request autoload vectors for all the pieces of your program, if you want to. If you use the easiest rule (described on the first page of this chapter) each global symbol referred to in a transfer-of-control statement closer to the root will have an autoload vector. But autoload vectors are only necessary when a transfer of control is made to a piece that is not currently in memory, so that it can be properly loaded. You can save four words for each unnecessary autoload vector you eliminate by using the autoload indicator only where it is really needed.

To understand how to request specific autoload vectors, you must understand how the Task Builder has stored the pieces of your program on disk, and how and when the appropriate pieces are loaded. This was discussed in Section 3.5.2; that information is reviewed here with a more complex example.

When you request overlays, the Task Builder constructs your executable program file such that pieces will be loaded in the most efficient manner. It does this by analyzing your ODL file. Pieces connected by a hyphen (-) are stored such that they will be loaded in one disk access. Pieces separated by a comma are stored such that they require a separate disk access for each piece.

For example, consider the overlay structure in Figure 5–3.

Figure 5–3: An Overlay Structure Without Autoload Vectors



This structure can be represented (including the FORTRAN library F4POTS.OLB) by the following ODL file. Note that only the structure is shown—no autoload vectors for the moment, although they will be needed.

```
          .ROOT CNTRL-LIBR-(AFCTR,BFCTR)
AFCTR:    .FCTR A0WLIB-(A1WLIB,A2WLIB-(A21WLB,A22WLIB)
BFCTR:    .FCTR B1-B2-B3-B4-B5-LIBR
A0WLIB:   .FCTR A0-LIBR
A1WLIB:   .FCTR A1-LIBR
A2WLIB:   .FCTR A2-LIBR
A21WLB:   .FCTR A1-LIBR
A22WLB:   .FCTR A22-LIBR
LIBR:     .FCTR LB:F4POTS/LB
          .END
```

5.4 Example of Specific Autoload Vector Requests

Now that you understand how autoload indicators apply to the various elements possible in an ODL file, return to the specific example in Figure 5–3 and the ODL file following it.

Suppose that CNTRL calls B3, which makes various calls to B1, B2, B4, and B5 before it returns control. CNTRL then calls A21, which calls A2 and A0. Control returns to A21, which returns control to CNTRL. CNTRL then calls A1 and A22.

Thus, you need apply the autoload indicator only to B3 in the "B" branch of the structure. In the "A" branches, you must supply an autoload indicator for A21, A1, and A22. You can accomplish this with the following ODL file:

```
          .ROOT CNTRL-LIBR-(AFCTR,BFCTR)
AFCTR:    .FCTR A0WLIB-(A1WLIB,A2WLIB-(A21WLB,A22WLIB)
BFCTR:    .FCTR B1-B2-*B3-B4-B5-LIBR
A0WLIB:   .FCTR A0-LIBR
A1WLIB:   .FCTR *A1-LIBR
A2WLIB:   .FCTR A2-LIBR
A21WLB:   .FCTR *A21-LIBR
A22WLB:   .FCTR *A22-LIBR
LIBR:     .FCTR LB:F4POTS/LB
          .END
```

5.5 If You Make a Mistake

If you make an error in placing asterisks, you will have problems within your program. Suppose you leave out an asterisk so that an autoload vector is not generated for a piece of your program. That piece will then be called with a direct reference, and if it is not actually in memory, control passes to whatever happens to be there. The Task Builder cannot diagnose the error at build time, because it does not attempt to analyze the sequence of calls in your program.

So, be very careful in requesting that the Task Builder generate autoload vectors for only some of the pieces making up your program.

Chapter 6

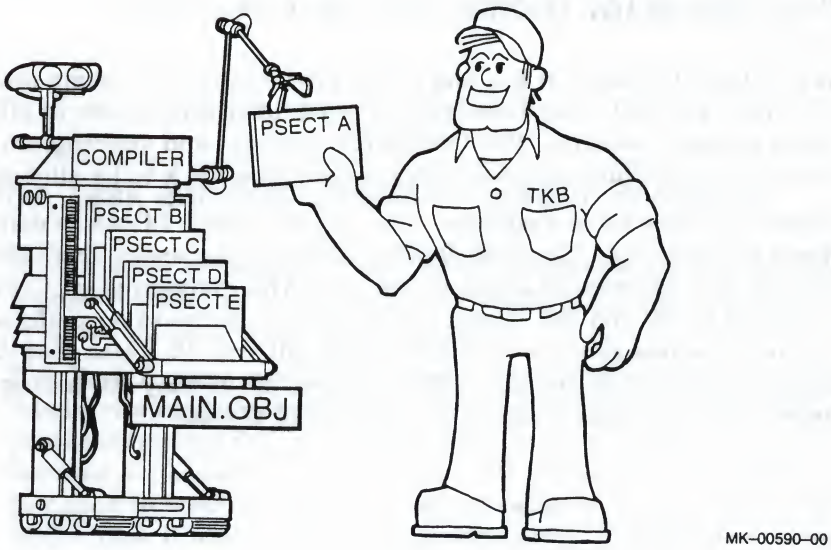
Working with Program Sections

So far, overlay techniques have been discussed in terms of units you are familiar with: files consisting of separately compiled or assembled programs or subprograms, or of library files. This chapter discusses the units these files consist of—the units the Task Builder actually works with—program sections.

6.1 What is a Program Section?

All the language translators produce program sections. With MACRO, you work directly with program sections. The .PSECT directive of the MACRO language lets you name and define exactly what goes into these units. With the higher-level languages, the compiler handles most aspects of generating and assigning attributes to program sections for you (Figure 6–1).

Figure 6–1: The Task Builder Works with Program Sections



The Task Builder allocates space differently for different parts of a program or subprogram, depending on certain attributes. With program sections, you can take full advantage of these attributes. A case where you would need program sections involves common areas, a programming feature of all languages.

Table 6–1 shows the program sections each file contains and their access codes and allocation codes.

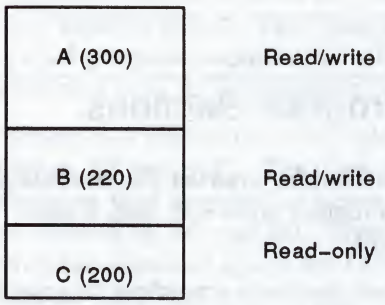
Table 6–1: Program Sections for IN1, IN2, AND IN3

File Name	Program Section Name	Access Code	Allocation Code	Size (octal)
IN1	B	RW	CON	100
	A	RW	OVR	300
	C	RO	CON	150
IN2	A	RW	OVR	250
	B	RW	CON	120
IN3	C	RO	CON	50

The Task Builder first determines the amount of space to allocate for each program section. Program section A appears in two files and has the overlay (OVR) attribute. The OVR attribute causes the Task Builder to allocate the largest of the two sizes, or 300 bytes, for A. Program section B appears twice and has the concatenate (CON) attribute. Thus, the total allocation for B is the sum of the lengths of each occurrence, or 220 bytes. The portion allocated for IN1 (100 bytes) is assigned the lowest addresses, since it appears first in the input file list. Program section C appears twice and, since it has the concatenate attribute, is allocated 200 bytes.

The Task Builder then groups the program sections according to their access codes, with the read/write sections being allocated the lowest addresses, followed by the read-only sections (see Figure 6–3).

Figure 6–3: Allocation of Program Sections for IN1, IN2, and IN3



MK-00592-00

The only other factor contributing to how the Task Builder allocates and orders program sections is whether the section consists of instructions or data. The Task Builder always allocates address space for a program section beginning on a word boundary. If the program section has the instruction (I) and concatenate (CON) attributes, the Task Builder appends the space so that each piece begins on a word boundary. (This eliminates the possibility of an odd-address transfer.) If the program section has the data (D) and concatenate (CON) attributes, however, and the space contributed by a piece ends on a byte rather than a word boundary, the space for the next piece is appended starting with the next byte.

6.4 The Task Builder's .PSECT Command

You can direct the placement of program sections at build time with the .PSECT command. Suppose, for example, that you wanted to place the common block COMB from the example in Figure 6–2 in the root section of the program. This would make the area accessible from both A0 and B0; they would be able to pass data in the common area, as desired. This could be accomplished with the following ODL commands:

```
.PSECT   COMB,RW,GBL,REL,OVR,D
.ROOT    ROOT-COMB-LIBR-*(A0WL-*(A1WL,A2WL-*(A1WL,A22WL)))
  .
  .
  .
.END
```

In the .PSECT command, you specify the program section name first, followed by its attributes in any order. The attributes shown here are typical for a common block: read/write (RW), global (GBL), relocatable (REL), overlaid (OVR), and data (D).

6.5 Using .NAME to Make a Data PSECT Autoloadable

You can construct an object file consisting only of data and make that file autoloadable. For example, suppose that, using MACRO, you constructed a file consisting of a program section containing error messages. Suppose further that you wanted to overlay this file, because it was needed only when a certain subprogram was running. Such overlaying can be accomplished, and is perhaps best explained by example.

Consider the subprogram ERDAT.OBJ, which processes an error value. If the value is 0, the subprogram calls a routine named ALRIT.OBJ. If the value is not 0, however, it displays one of the error messages contained in the file MSG.OBJ, consisting of a program section with the D attribute, using, say, the MACRO language .ASCII command to define each particular error message.

There is no reason why ALRIT and MSG must be in memory at the same time. You can overlay ALRIT and MSG, by using the .NAME directive to define a name and attributes for the file MSG. For example:

```
        .ROOT   MAIN-*(OTHER,ERMSG-(ALRIT,MSG))
        .NAME   MESAGE,GBL
MSGF:   .FCTR   MESAGE-MSG
        .END
```

That is, you must include a .NAME command defining a global name (MESAGE in this case) for the data file (MSG in this case). The Task Builder generates the global symbol MESAGE and enters it into the symbol table for segment MESAGE. Since this segment is included for the generation of autoload vectors, an autoload vector is created for the segment referred to by the global symbol MESAGE. Because it consists only of a program section with the data (D) attribute, however, the Task Builder constructs a special autoload vector. The last word, normally an "entry point address" for an autoloaded segment, instead refers to the symbol $$RTS (generated by the Task Builder, and containing a simple return instruction).

Following this listing of program sections and their memory allocations is a list of global symbols defined or used in each segment. Note that the only global symbols recognizable from the source program are USER, INTRO, CRUNCH, and CHATR, assigned to the entry point of each routine by the compiler. The rest have been assigned by the compiler, or belong to the library routines that have been inserted into the segment.

The last page of the listing contains Task Builder statistics on the build. These statistics are mainly useful in analyzing Task Builder performance on your system, as described in Appendix E.

6.6.1 Source for Program USER

```
10          CALL INTRO(A1%,B1%)
20          CALL CRUNCH(A1%,B1%,SUMM%,PRODUCT%,DIFFER%)
30          CALL CHATR(A1%,B1%,SUMM%,PRODUCT%,DIFFER%)
40          END
```

6.6.2 Source for Subprogram INTRO

```
10          SUB INTRO(AA%,BA%)
20          INPUT "INPUT TWO NUMBERS";AA%,BA%
30          SUBEND
```

6.6.3 Source for Subprogram CRUNCH

```
10          SUB CRUNCH(AB%,BB%,CA%,CB%,CC%)
20          CA% = AB% + BB%
30          CB% = AB% * BB%
40          CC% = AB% - BB%
50          SUBEND
```

6.6.4 Source for Subprogram CHATR

```
10          SUB CHATR(AC%,BC%,CA%,CB%,CC%)
20          PRINT "THE SUM OF ";AC%;" AND ";BC%;" IS ";CA%
30          PRINT "THE PRODUCT OF ";AC%;" AND ";BC%;" IS ";CB%
40          PRINT "THE DIFFERENCE OF ";AC%;" AND ";BC%;" IS ";CC%
50          SUBEND
```

6.6.5 Overlay Description File FRED.ODL

```
            .ROOT USWL-*(INTRWL,CRUNWL,CHATWL)
USWL:       .FCTR USER-LIBR
INTRWL:     .FCTR INTRO-LIBR
CRUNWL:     .FCTR CRUNCH-LIBR
CHATWL:     .FCTR CHATR-LIBR
LIBR:       .FCTR LB:BP2OTS/LB
            .END
```

6.6.6 Task Builder Command File

```
USER,USER=FRED/MP
UNITS=12
ASG=SY:5:6:7:8:9:10:11:12
EXTTSK=512
//
```

6.6.7 Task Builder Listing

```
USER.TSK   Memory allocation map  TKB 08.006      Page 1
                    15-MAY-90   14:00

Partition name : GEN
Identification : 000708
Task  UIC      : [30,21]
Stack    limits: 001000 001777 001000 00512.
PRG xfr address: 016030
Total address windows: 1.
Task   extension   : 512. words
Task  image  size  : 6304. words
Total  task  size  : 6816. Words
Task address limits: 000000 030453
R-W disk blk limits: 000002 000041 000040 00032.


USER.TSK Overlay description:

Base     Top        Length
----     ---        ------
000000   020773  020774  08700.    USER
020774   030453  007460  03888.      INTRO
020774   021427  000434  00284.      CRUNCH
020774   026023  005030  02584.      CHATR

USER.TSK   Memory allocation map  TKB 08.006      Page 2
USER                 15-MAY-90   14:00


*** Root segment: USER


R/W mem  limits: 000000 020773 020774 08700.
Disk blk limits: 000002 000022 000021 00017.
```

```
$CLFQB 003720-R  $FPHSK 015172-R  $IOERV 006710-R  $SETSC 006236-R
$CLOSR 013034-R  $FPUER 014614-R  $IOTST 004124-R  $SPEC  003224-R
$CLSAL 013164-R  $FRCER 014634-R  $MEMPR 014630-R  $STCRE 010116-R
$CLSFQ 004106-R  $FSS   004030-R  $MEMP1 011674-R  $STCRX 010122-R
$CLSHD 013152-R  $FSSCN 013600-R  $MNIUS 010456-R  $STMOV 010150-R
$CLSTK 010220-R  $FSSCZ 013602-R  $MNSUB 010522-R  $STMVX 010162-R
$CLXRB 003742-R  $GSACM 013204-R  $MREST 011004-R  $SYSHD 011274-R
$CNVIA 015456-R  $GTPTN 007734-R  $NOREX 011520-R  $VALDC 015710-R
$DATRC 006626-R  $GTPTR 010022-R  $ODDAD 014624-R  $VALID 015672-R
$DATRS 006604-R  $GTROM 007520-R  $ODDA1 011654-R  $VREAD 003252-R
$DOIT  012476-R  $GTR01 014142-R  $ONERG 011126-R  $XWRT  004002-R
$DOIT1 012520-R  $GTR23 014246-R  $OTSVA 016724-R  $$MAXC 000017
$ERROR 014614-R  $GTSTN 007724-R  $POMSK 007440-R  ..B2TK 015640-R
$ERRT1 010660-R  $GTSTR 010010-R  $PROCT 015176-R  ..CRLF 015576-R
$ERTXT 011532-R  $ICIO0 016244-R  $POMSK 007242-R  ..PMD  015574-R
$EXTSP 007666-R  $ICJMP 015754-R  $PSMS2 007234-R  ..PTXT 015602-R
$FLSAL 014426-R  $ICJM1 015754-R  $PSMS3 007226-R  ..RSTT 015554-R
$FLSFR 014436-R  $INITM 005274-R  $RELCB 014730-R  ..SVFQ 004136-R
$FLSNL 014454-R  $INITS 002270-R  $REQCB 015032-R
$FLUSH 014464-R  $INTCM 006432-R  $RSU2  010644-R
$FPASX 015174-R  $IOERS 006724-R  $SAVRE 016004-R

USER.TSK   Memory allocation map  TKB 08.006      Page 5
INTRO                 15-MAY-90   14:00


*** Segment: INTRO

R/W mem  limits: 020774 030453 007460 03888.
Disk blk limits: 000023 000032 000010 00008.

Memory allocation synopsis:

Section                                          Title  Ident  File
-------                                          -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)     020774 000000 00000.
BP2OTS:(RO,I,LCL,REL,CON)      020774 007272 03770.
                               020774 000422 00274. $ICINI 23CM   BP2OTS.OLB
                               021416 002004 01028. $ICRED 53CM   BP2OTS.OLB
                               023422 000656 00430. $ICWRT 04CM   BP2OTS.OLB
                               024300 000642 00418. $STMOS 16CM   BP2OTS.OLB
                               025142 002404 01284. $ECONV 24CM   BP2OTS.OLB
                               027546 000226 00150. $ICFNS 11RE   BP2OTS.OLB
                               027774 000202 00130. $STLSS 08CM   BP2OTS.OLB
                               030176 000070 00056. $STFN1 06CM   BP2OTS.OLB
$ARRAY:(RW,D,LCL,REL,CON)      030266 000000 00000.
                               030266 000000 00000. INTRO  000708 INTRO.OBJ
$CODE :(RO,I,LCL,REL,CON)      030266 000134 00092.
                               030266 000134 00092. INTRO  000708 INTRO.OBJ
$FLAGR:(RW,D,GBL,REL,CON)      016234 000000 00000.
                               016234 000000 00000. INTRO  000708 INTRO.OBJ
$FLAGS:(RW,D,GBL,REL,CON)      016234 000010 00008.
                               016236 000002 00002. INTRO  000708 INTRO.OBJ
$FLAGT:(RW,D,GBL,REL,CON)      016244 000000 00000.
                               016244 000000 00000. INTRO  000708 INTRO.OBJ
$IDATA:(RW,D,LCL,REL,CON)      030422 000004 00004.
                               030422 000004 00004. INTRO  000708 INTRO.OBJ
$PDATA:(RO,D,LCL,REL,CON)      030426 000026 00022.
                               030426 000026 00022. INTRO  000708 INTRO.OBJ
$STRNG:(RW,D,LCL,REL,CON)      030454 000000 00000.
                               030454 000000 00000. INTRO  000708 INTRO.OBJ
$TDATA:(RW,D,LCL,REL,CON)      030454 000000 00000.
                               030454 000000 00000. INTRO  000708 INTRO.OBJ
$$ALVC:(RO,I,LCL,REL,CON)      030454 000000 00000.
$$RTS :(RO,I,GBL,REL,OVR)      020710 000002 00002.
```

```
Global symbols:

ASC$    030234-R  IPT$    021246-R  LIT$    021166-R  MOS$AP 024516-R
BUF$    027752-R  IRD$    021002-R  LMA$1   030070-R  MOS$AS 024300-R
CCP$    027630-R  IVF$A   021416-R  LSS$AA  030000-R  MOS$MA 024622-R
CHR$    030256-R  IVI$A   021524-R  LSS$AM  030020-R  MOS$MM 024756-R
III$    021226-R  IVS$A   021564-R  LSS$AP  027774-R  MOS$MP 024716-R
IIN$    021120-R  LAM$1   030022-R  LSS$MA  030050-R  MOS$MS 024340-R
ILI$    021206-R  LAM$2   030026-R  LSS$PA  030044-R  MOS$PA 024626-R
ILS$    021102-R  LEN$    030224-R  MOS$AA  024556-R  MOS$PM 024510-R
INTRO   030266-R  LIS$    021064-R  MOS$AM  024502-R  MOS$PP 024712-R

USER.TSK   Memory allocation map  TKB 08.006      Page 6
INTRO                  15-MAY-90   14:00


MOS$PS 024330-R  NSS$PA 030156-R  STS$    027616-R  $FTOAX 026256-R
MOS$SA 024554-R  PVD$SI 023422-R  TAB$    027546-R  $INPTT 022170-R
MOS$SM 024444-R  PVF$SI 023432-R  WAT$    027732-R  $ISETP 022034-R
MOS$SP 024374-R  PVI$SI 023572-R  $ATOD   025142-R  $I4ER  021504-R
MOS$SS 024360-R  PVS$AI 023442-R  $CRLF   023724-R  $POS   027646-R
MOS$01 025056-R  RCT$   027604-R  $DMAXD  027544-R  $PRNSP 024060-R
MS1$01 024334-R  RST$   020774-R  $DTOA   026320-R  $PRNTL 024076-R
NMA$1  030172-R  SPC$   030176-R  $DTOAX  026324-R  $SETUP 023772-R
NSS$AA 030136-R  SPC$01 030176-R  $FMAXD  027542-R
NSS$MA 030162-R  STR$   030204-R  $FTOA   026252-R

USER.TSK   Memory allocation map  TKB 08.006      Page 7
CRUNCH                 15-MAY-90   14:00


*** Segment: CRUNCH

R/W mem  limits: 020774 021427 000434 00284.
Disk blk limits: 000033 000033 000001 00001.

Memory allocation synopsis:

Section                                          Title  Ident  File
-------                                          -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)     020774 000000 00000.
BP2OTS:(RO,I,LCL,REL,CON)      020774 000234 00156.
                               020774 000046 00038. $JPADD 01CM   BP2OTS.OLB
                               021042 000074 00060. $JPMOV 02CM   BP2OTS.OLB
                               021136 000024 00020  $JMUL  01CM   BP2OTS.OLB
                               021162 000046 00038. $JPSUB 01CM   BP2OTS.OLB
$ARRAY:(RW,D,LCL,REL,CON)      021230 000000 00000.
                               021230 000000 00000. CRUNCH 000802 CRUNCH.OBJ
$CODE :(RO,I,LCL,REL,CON)      021230 000154 00108.
                               021230 000154 00108. CRUNCH 000802 CRUNCH.OBJ
$FLAGR:(RW,D,GBL,REL,CON)      016234 000000 00000.
                               016234 000000 00000. CRUNCH 000802 CRUNCH.OBJ
$FLAGS:(RW,D,GBL,REL,CON)      016234 000010 00008.
                               016240 000002 00002. CRUNCH 000802 CRUNCH.OBJ
$FLAGT:(RW,D,GBL,REL,CON)      016244 000000 00000.
                               016244 000000 00000. CRUNCH 000802 CRUNCH.OBJ
$IDATA:(RW,D,LCL,REL,CON)      021404 000012 00010.
                               021404 000012 00010. CRUNCH 000802 CRUNCH.OBJ
$PDATA:(RO,D,LCL,REL,CON)      021416 000012 00010.
                               021416 000012 00010. CRUNCH 000802 CRUNCH.OBJ
$STRNG:(RW,D,LCL,REL,CON)      021430 000000 00000.
                               021430 000000 00000. CRUNCH 000802 CRUNCH.OBJ
$TDATA:(RW,D,LCL,REL,CON)      021430 000000 00000.
                               021430 000000 00000. CRUNCH 000802 CRUNCH.OBJ
$$ALVC:(RO,I,LCL,REL,CON)      021430 000000 00000.
$$RTS :(RO,I,GBL,REL,OVR)      020710 000002 00002.
```

Part III
System Aspects

For example:

```
RUN $TKB
TKB>MYLIB/-HD,,MYLIB=CODE1,CODE2,CODE3
TKB>/
ENTER OPTIONS:
TKB>PAR=MYLIB:140000
TKB>STACK=0
TKB>//
```

The area above would always have to be linked as follows:

```
RUN $TKB
TKB>PROG=PROG
TKB>/
ENTER OPTIONS:
TKB>LIBR=MYLIB:RO:6
TKB>//
```

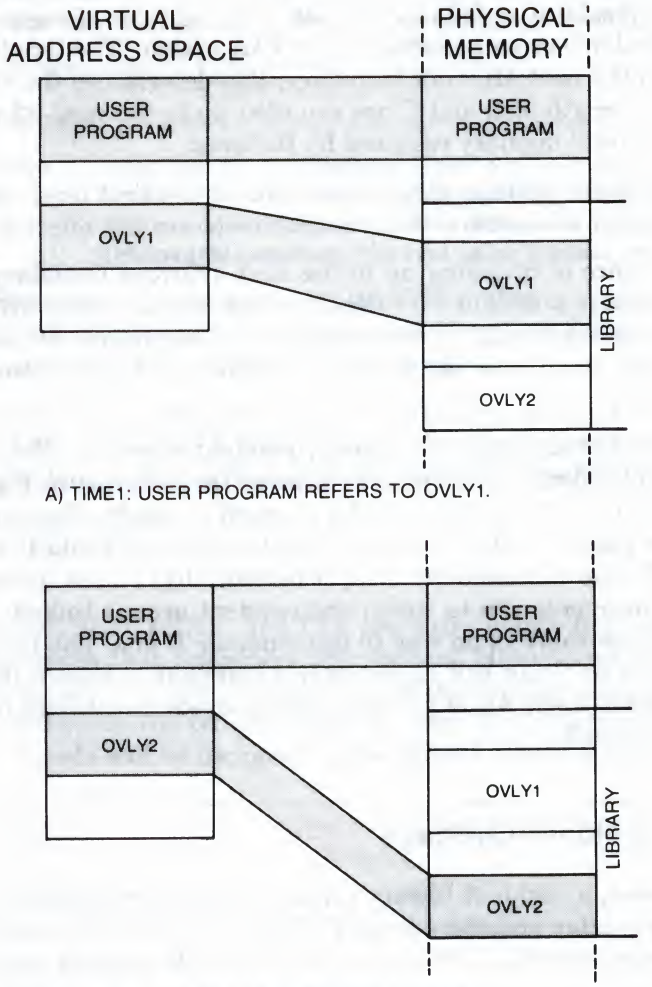
That is, since it was built to begin at location 140000, it must always be linked using APR 6. Note also that MYLIB.TSK and MYLIB.STB must be on the device in the account denoted by the system logical LB:, because LIBR was used rather than the RESLIB option.

7.4 Resident Areas with Memory-Resident Overlays

The Task Builder lets you construct what are called "memory-resident overlays" for resident areas. Memory-resident overlay segments are loaded from disk when your program is loaded; thereafter, they reside in memory as long as any other program in memory is using them. Memory-resident overlays share virtual address space, just as the disk-resident overlays that we have discussed. Unlike disk-resident overlays, memory-resident overlays do not share actual memory. Instead, they reside in separate areas of actual memory. The virtual address space is shared by the mapping technique described in Chapter 2.

For example, consider Figure 7–1. At time 1, the job area in virtual address space contains OVLY1, one segment of a resident area with memory-resident overlays. At time 2, the job area in virtual address space contains OVLY2, the other segment of the resident area with memory-resident overlays. Both segments OVLY1 and OVLY2 reside in physical memory; they are mapped into the virtual address space at different times.

Figure 7–1: Memory-Resident Overlays



7.4.1 Specifying Memory-Resident Overlays

You can use many of the same techniques in doing memory-resident overlays for resident areas as you use for disk-resident overlays. As with disk-resident overlays, the branches of an overlay tree must be logically independent (see Section 3.6). In the example in Figure 7–1, OVLY1 cannot call or refer to data in OVLY2, or vice versa.

In the ODL file, use an exclamation point to specify memory-resident overlay segments. Memory-resident overlay segments are indicated by placing an exclamation point immediately before the left parenthesis enclosing the desired segments. For example:

```
.ROOT A-!(B,C)
```

7.5.7 Sample Vector Table Code

The code below illustrates part of a sample vector table.

```
.OPEN:: MOV     #30,-(SP)           ;PUT OFFSET INTO USER PROGRAM
                                    ;JUMP TABLE ON THE STACK
        BR      DISPAT              ;JOIN COMMON DISPATCH
            .
            .
            .
DISPAT: MOV     R0,-(SP)            ;SAVE REGISTER
        MOV     @#.FSRPT,R0         ;GET POINTER TO DATA AREA
        ADD     A.JUMP(R0),2(SP)    ;ADD VECTOR BASE TO OFFSET
        MOV     (SP)+,R0            ;RESTORE REGISTER
        MOV     @(SP)+,-(SP)        ;PICK UP ADDRESS OF TARGET
        JMP     @(SP)+              ;AND TRANSFER TO TARGET
```

In the example above, the code at .OPEN pushes the known offset into the jump table (30) onto the stack and transfers control to dispatch code, common for all the revectored entry points. The code at DISPAT:

1. Pushes the contents of R0 onto the stack, to save it.
2. Moves the address of the base of the data area into R0.
3. Adds the base address of the jump table to the index onto the stack.
4. Restores R0.
5. Puts the address of the desired routine's autoload vector onto the stack.
6. Jumps to the autoload vector for the desired routine (.OPEN).

7.5.8 GBLXCL and GBLINC Options

In the preceding example, notice that both the calling library and the called library contain an entry point named .OPEN. You must exclude global symbols for such revectored entry points from the calling library's symbol table (.STB) file, or the Task Builder has a hard time figuring out which one to use when the libraries are referenced during a user program's build. To do this, use the GBLXCL option when you are building the calling library.

Another aspect of building the calling library is ensuring that the needed jump tables are built into the user program when the calling library is referenced there. This involves placing the code for the jump tables in the system library, or a library always referenced by the user program, and ensuring that such code is always included in the user program when the calling library is referred to in the user program. You use the GBLINC option in the calling library to do this.

The example below shows the build for the "calling library," called US1CLS.

```
RUN $TKB
TKB>US1CLS/-HD,US1CLS/CR/-SP/MA,US1CLS=US1LIB.OBJ
TKB>LB:SYSLIB/LB:FCSVEC
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=US1CLS:140000:4000
TKB>GBLINC=.FCSJT
TKB>GBLXCL=.OPEN
TKB>//
```

The example above shows the vector table (FCSVEC) as a module in the system library. Building such a vector table as part of a commonly used library, such as SYSLIB, makes it easier for more than one library to access the called library.

The GBLINC option shown forces the Task Builder to add a global reference entry for .FCSJT in the library's .STB file. This ensures that the Task Builder links the jump table modules required by the library into the user program. These modules should be in the system library, or in a library always referenced by the user program. Thus, this forced loading mechanism is invisible to the user.

7.6 FORTRAN Virtual Arrays

The FORTRAN-77 programming system automatically provides a virtual array mechanism for up to 255K words of data. The FORTRAN user can generate a unique dynamic region for data by making the proper declarations within the control of the FORTRAN language. Refer to the FORTRAN-77 *User's Guide* and other language manuals for more information.

NOTE

The size of the virtual arrays created may be limited by the dynamic region limit. See the *RSTS/E System Manager's Guide*.

If a FORTRAN program requires more memory-based data than the virtual arrays can provide, use virtual program selectors.

Use the following FORTRAN-77 statement to invoke the automatic virtual array mechanism:

```
VIRTUAL var(index), var2(index2),...
```

The FORTRAN compiler informs the Task Builder of the required size of the virtual array area. The TKB instructs the RSTS monitor to create the proper-sized dynamic region automatically. In RSX-11M-PLUS, this offset size is additional memory allocated to the task partition. In RSTS/E, it is the size of the dynamic region that will hold the virtual array data. This unnamed dynamic region will be created when the task is loaded at run-time and will always have the region ID value of zero. If not enough memory to allocate the dynamic region exists (see the *RSTS/E System Manager's Guide*), the task will not start. The FORTRAN language will automatically map the region as needed to access the data.

You can use the automatic region creation feature in MACRO-11 programs by including one of the following:

- The program of a .PSECT of the form:

```
        .psect NAME,D
varnam::
        UNORG
```

where:

name is a unique section name

7.7.2 Building a Program That Uses a Virtual Program Section

Example 7-2 shows the FORTRAN source file for a task named VSECT.FTN. The example illustrates the use of the ALSCT FORTRAN subroutine, and uses the automatic region for both VARRAY and IARRAY. When you build, install, and run VSECT2, it allocates the mapped array area in a dynamic region, creates a 4K-word window, and maps to the area through the window. ALSCT then initializes the area and prompts for an array subscript at your terminal, as follows:

```
SUBSCRIPT?
```

When you input a subscript, it responds with ELEMENT= and the contents of the array element for the subscript you typed. VSECT2 continues to prompt until you press Ctrl/Z at which time it exits.

Once you have compiled VSECT2, you can build it with the following TKB command sequence:

```
TKB @VSECT2
```

Example 7–1: VSECT2.CMD

```
>
;           VSECT2.CMD
SY:VSECT2/ID/FP,vsect2=vsect2.odl/MP
UNITS =12
ASG   =SY:7:8:9:10:11:12
EXTTSK=512
MAXBUF=512
WNDWS=2
LIBR=RMSRES:RO:6:0
VSECT=MARRAY:160000:20000:2000
//
```

Note the following in Example 7-1:

- The /ID allows the use of the D-APRs where the RMSRES is placed.
- The RMSRES is at APR 6 and the APRs are unprotected in the D-space.
- The VSECT option has a physical length of 4K words which will be added to the length needed for the VARRAY( ).
- The WNDWS option is set allow for both types of arrays used.

This command file sequence directs TKB to create a task image file named VSECT.TSK and a map file named VSECT2.MAP. (See Example 7-3.)

The WNDS option directs TKB to add a window block to the header in the task image. The Executive initializes this window block when it processes the mapping directives within the task. This allows two programmed address windows (CRAW$) to be created in this region.

The VSECT option directs TKB to establish for the program section named MARRAY a base address of 160000 (APR 7) and a length of $(20000)_8$ bytes (4K words). The program section MARRAY is defined within the task through the FORTRAN COMMON statement. The VSECT option also specifies that TKB is to

allocate 200 64-byte blocks of physical memory in the task's mapped array area in the automatic dynamic region. For more information on the switches and options used in this example, refer to Chapters 11 and 12, respectively.)

This command sequence results in the map shown in Example 7-3.

The DCL link for FORTRAN-77 also creates the following ODL file:

```
;        VSECT2.ODL
; TKB .ODL file created by DCL LINK  V10.0-F    09-May-90   09:49 AM
@LB:RMS11M.ODL
@LB:RMSRLX.ODL
ROOT$$: .FCTR   OTSROT-RMSROT
        .NAME   LBR$$
        .ROOT   A0$,OTSALL,RMSALL
A0$:    .FCTR   SY:VSECT2-ROOT$$-LBR$$
.END
```

Example 7–2: Source Listing for VSECT2.FTN

```
c
c       vsect2.ftn
c
        VIRTUAL VARRAY(1500)
        integer *2 sub,irdb(9),iwdb(11),iwdb2(11),dsw
        integer *2 IARRAY(4096)
        common /MARRAY/IARRAY
c       Set up the window for the manual virtual array.
        iwdb(1)="3400                !use apr 7 for window
        iwdb(3)=128                  !window size=128*32 words=4096 (1 apr)
        iwdb(5)=0                    !offset =0
        iwdb(7)="422                 !status=WS.64B!WS.WRT!WS.UDS
        iwdb(10)= 128                !size to allocate =4Kw
c       Set up the window for the automatic virtual array.
c       This window will not actually be used--it is just a place-holder.
        iwdb2(1)="3400               !use apr 7 for window
        iwdb2(3)=96                  !window size=96*32 words=3072
        iwdb2(5)=0                   !offset =0
        iwdb2(7)="422                !status=WS.64B!WS.WRT!WS.UDS
        iwdb2(10)= 96                !size to allocate =3Kw
        irdb(2)=340                  !region size expected (total)
        irdb(9)=0                    !assure link is zero for first call
c
c       Allocate 3K mapped area for the VARRAY (not to be used)
c
        call ALSCT(irdb,iwdb2,dsw)
        if (dsw .ne. 1) goto 100
c
c       Allocate 4K mapped array
c
        call ALSCT(irdb,iwdb,dsw)
        if (dsw .ne. 1) goto 100
c
c       Create a 4K window
c
        call CRAW(iwdb,dsw)
        if (dsw .ne. 1) goto 200
c
c       Map the array into virtual space
c
        call MAP(iwdb,dsw)
        if (dsw .ne. 1) goto 300
```

(continued on next page)

Example 7–2 (Cont.): Source Listing for VSECT2.FTN

```
        do 10 i=1,4096
10      IARRAY(i)=i
c
c       Mapped array initialized
c
30      write (5,35)
35      format ('subscript?')
        read (5,40, END=1000) sub
40      format (i7)
        write(5,60) IARRAY(sub)
60      format (' element = ',I7)
        goto 30
c
c       error entries
c
100     write (5,101)dsw
101     format (' error from ALSCT, error= ',I7)
        goto 1000
200     write (5,201)dsw
201     format (' error from CRAW, error= ',I7)
        goto 1000
200     write (5,201)dsw
201     format (' error from CRAW, error= ',I7)
        goto 1000
300     write(5,301)dsw
301     format (' error from mapping, error= ',I7)
        goto 1000
1000    call exit
        end
```

Example 7–3: Task Builder Map (Edited) for VSECT2.TSK

```
VSECT2.TSK   Memory allocation map  TKB M43.00      Page 1
                    18-MAY-90   09:19

Partition name : GEN
Identification : 18MAY
Task  PPN      : [216,1]
Stack    limits: 001000 001777 001000 00512.
PRG xfr address: 002424
Task attributes: ID
Total address windows: 7.
Mapped  array  area: 35840. words
Task   extension   : 512. words
Task  image  size  : 6720. words, I-Space
                     5376. words, D-Space
Total  task  size  : 6720. words, I-Space
                     41728. words, D-Space
Task Address limits: 000000 032133 I-Space
                     000000 024703 D-Space
R-W disk blk limits: 000002 000142 000141 00097.

Memory allocation synopsis:

Section                                          Title  Ident  File
-------                                          -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)    001000 000000 00000.
MARRAY:(RW,D,GBL,REL,OVR)    160000 020000 08192.
                             160000 020000 08192. .MAIN. 18MAY  VSECT.OBJ
                  .
                  .
$VARS :(RW,D,LCL,REL,CON)    002262 000104 00068.
                             002262 000104 00068. .MAIN. 18MAY
VSECT.OBJ
$VIRT :(RW,D,GBL,REL,CON)    002000 000140 00096.
                             002000 000136 00094. .MAIN. 18MAY
VSECT.OBJ
$$ALER:(RO,I,LCL,REL,CON)    003220 000040 00032.
                  .
                  .
```

If the length to map is not specified (high bit of R0 is 0), the length is assumed to be the length given in the last PLAS mapping (either the value in XRB at W.NLEN is CRAW$ or MAP$, or FQBUFL if CRAFQ or MAPFQ).

If the length is specified (high bit of R0 is 1), then that length is mapped. If a length of 0 is given in R2, then the smaller of either the last .PLAS mapping length or the remaining length in the library between the offset and the library top is mapped. This handling is identical to that for the .PLAS mapping for the same case.

Note the difference between the RSTS/E and RSX-11M-PLUS versions is that User D-space must remain mapped to user's low core (APR0) at all times in RSTS/E. RSX-11M-PLUS does not have such a restriction.

Note also that the speed of fast-mapping is affected by the parameter values. Not specifying the length to map is the fastest and specifying a length equal to zero is the slowest. The improvement over the .PLAS directive is in the range of six to twelve times faster. The amount of improvement depends on the amount of remapping done by the task.

7.9.2.1 Calling Sequence

R0	Parameter 1	See Table 7-2
R1	Parameter 2	Offset field
R2	Parameter 3	Optional length specification

7.9.2.2 Returned Data

R0	Status	IS.SUC if successful
		IE.ITS or IE.ALG if failure
R2	Length mapped	In slivers if successful
R1,R3		Indeterminate

7.9.2.3 Programming Examples

Changing only window offset in I-space:

```
MOV    #40,R0       ;window at APR 4 in I-space
MOV    #200,R1      ;offset from base of library is 4K words
                    ;200 slivers * 32 words/sliver
                    ;R2 is not needed

IOT                 ;issue fast map call
TST    R0           ;successful?
BMI    ERROR        ;no
```

Changing window offset and actual length specified:

```
MOV    #100150,R0   ;window at APR 5 in D-space
MOV    #100,R1      ;offset 2K words from library base
MOV    #500,R2      ;specified length = 10K words
                    ;this will use APRs 5, 6, & 7
IOT                 ;issue fast map call
TST    R0           ;successful?
BMI    ERROR        ;no
```

Changing window offset and defaulted length specified:

```
MOV    #100140,R0   ;window base at APR 4 in D-space
CLR    R1           ;offset=0, window begins at library base
CLR    R2           ;force calculation to FIRQB+12(octal) or
                    ;remaining size of library
IOT                 ;issue fast map call
TST    R0           ;successful?
BMI    ERROR        ;no
MOV    R2,MAPLEN    ;copy of size actually mapped, in slivers
```

In the above text and examples, the term "library" means a resident library or dynamic data region.

The following are additional notes on the use of the fast map facility:

- Fast-mapping cannot be used if ODT is included in the task build, because the IOT instruction is used differenctly for both facilities.
- Fast map cannot be used for mapping supervisor modes or called from supervisor mode.
- Fast-mapping can be turned on and off in the program by doing a .SET or .CLEAR system call to RSTS/E (see the *RSTS/E System Directives Manual* for details of usage). Note that RSX-11M-PLUS does not support this programmable control.

The conventional task in an I- and D-space system uses both sets of APRs. However, the relocation addresses in both I- and D-space APRs are identical.

An I- and D-space task can use both I- and D-space APRs independently; that is, APRs used in this way are not overmapped. Because of this, the task can use eight D-space APRs to access and use data, and eight I-space APRs to access and execute instructions. Using 16 APRs allows the I- and D-space task to access a total of 64K words of physical memory at one time.

Table 8–1 summarizes APR mapping for various combinations of I- and D-space tasks and I- and D-space systems.

Table 8–1: Mapping Comparison Summary

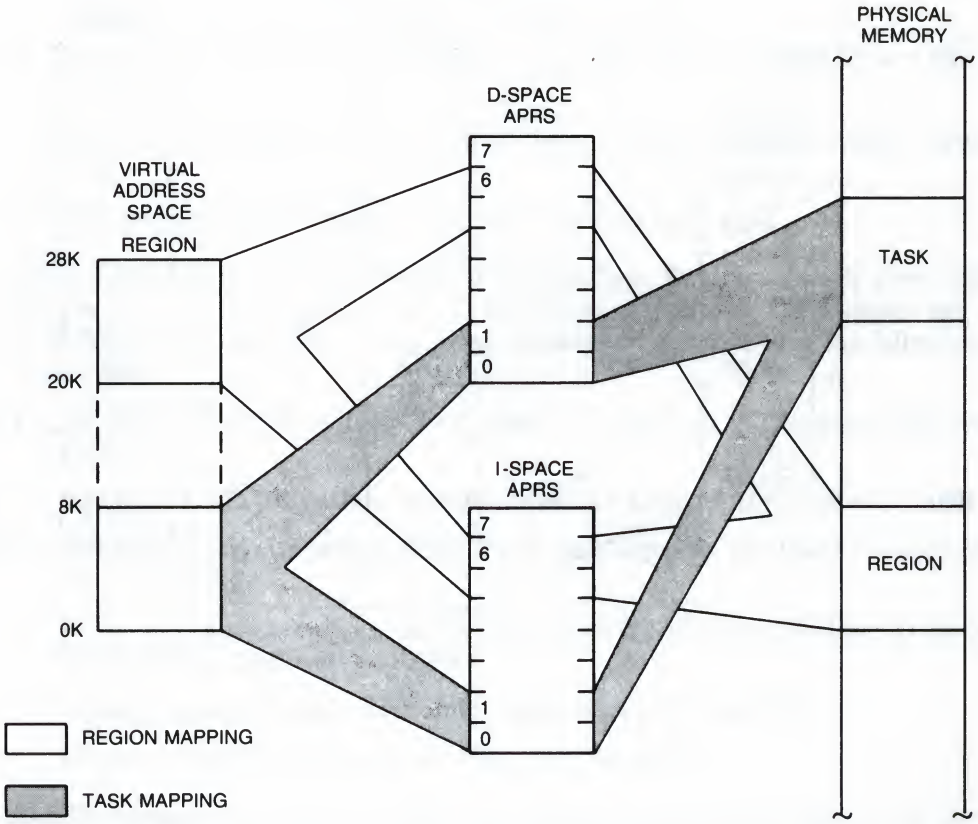
I/D Task	I/D System	Mapping Summary
No	Yes	I-space APRs and D-space APRs contain the same relocation addresses.
Yes	Yes	I-space APRs map instruction space. D-space APRs map data space.
Yes	No	Missing special feature error at run time.

8.4 Conventional Task Mapping

Conventional tasks map their virtual addresses to their logical addresses through both I-space and D-space APRs. That is, the Task Builder does not separate instruction space or data space, and the system does not differentiate the spaces except by the logic inherent in the task. Therefore, the task must map to its logical address space by both sets of APRs, which are overmapped.

Figure 8–1 shows an 8K-word task that does not use I- and D-space, and that is built against an 8K-word resident library.

Figure 8–1: Conventional Task Linked to a Region in I- and D-Space System



MK-01678-00

8.5 I- and D-Space Task Mapping

Figure 8–2 shows an 8K-word I- and D-space task. The Task Builder separated the data and instructions in this task. Because of the way the Task Builder processes task space, the task header must physically reside at the beginning of the task in I-space. The Task Builder puts the header that the monitor uses for task control in D-space. The task's stack is also in D-space.

The task in Figure 8–2 uses two APRs because of its size (8K-word). D-space APR 0 maps the task's header and stack and part of D-space.

Chapter 9

Supervisor-Mode Library

A supervisor-mode library is a resident library that doubles a user task's virtual address space by mapping the instruction space of the processor's supervisor mode.

A call from within a user task to a subroutine within a supervisor-mode library causes the processor to switch from user mode to supervisor mode. The user task transfers control to a mode-switching vector that the Task Builder includes within the task. After performing the mode switch, the vector transfers control to the called subroutine within the supervisor-mode library. When the library routine finishes executing, it transfers control to a completion routine within the library, which switches the processor back to user mode. The user task continues executing with the processor in user mode at the return address on the stack.

9.1 Mode-Switching Vectors

In a task that links to a supervisor-mode library, TKB includes a 4-word, mode-switching vector in the user task's address space for each entry point referred to in a subroutine in the library.

The following example shows the contents of a mode-switching vector:

```
MOV #COMPLETION-ROUTINE,-(SP)
CSM #SUPERVISOR-MODE-ROUTINE ADDRESS
```

NOTE

When switching from user mode to supervisor mode, all registers of the referencing task are preserved. All condition codes in the PSW are saved on the stack before they are cleared and must be restored by the completion routine.

Example 9–3 (Cont.): Completion Routine $CMPCS from SYSLIB.OLB

```
; A routine address of 0 is special-cased to support return to
; supervisor mode from a user mode debugging aid (ODT).  In this
; case stack is the following:
;
;          (SP)   zero
;         2(SP)   PC from CSM to be discarded
;         4(SP)   PSW from CSM to be discarded
;         6(SP)   Super mode PC supplied by debugger
;        10(SP)   Super mode PSW supplied by debugger
;
; To allow positioning at virtual zero, this code must be in the
; blank PSECT which is first in the TKB's PSECT ordering.

        .PSECT
        .ENABL  LSB

; Debugger return to super mode entry. Must start at virtual zero

        CMP     (SP)+,(SP)+     ; Clean off PSW and PC from CSM

;
: **-$SRTI-SUPER mode RTI
;
; This entry point performs the necessary stack management to
; allow an RTI from super mode to either super mode or user mode.
; In this case, the stack is the following:
;
;          (SP)   Super mode PC
;         2(SP)   Super mode PSW

$SRTI:: TST     2(SP)           ; Returning to user mode?
        BR      70$             ; Join common code
; CSM transfer address, this word must be at virtual 10 in super
; mode

        .WORD   CSMSVR          ; CSM dispatcher entry

; Dispatch CSM entry

CSMSVR: MOV     6(SP),2(SP)     ; Set completion routine address for RETURN
        JMP     @(SP)+          ; Transfer to super mode library routine

;
; **-$CMPAL-Completion routine which sets up NZVC in the PSW
;
; Copy all condition codes to stacked PSW.  Current stack:
;
;          (SP)   PSW with condition codes cleared
;         2(SP)   Completion routine address (to be discarded)
;         4(SP)   Return address
;
```

(continued on next page)

Example 9–3 (Cont.): Completion Routine $CMPCS from SYSLIB.OLB

```
$CMPAL::BPL     40$             ;
        BNE     20$             ;
        BVC     10$             ;
        BIS     #16,(SP)        ; Set NZV
        BR      $CMPCS          ;
10$:    BIS     #14,(SP)        ; Set NZ
        BR      $CMPCS          ;
20$:    BVC     30$             ;
        BIS     #12,(SP)        ; Set NV
        BR      $CMPCS          ;
30$:    BIS     #10,(SP)        ; Set N
        BR      $CMPCS          ;
40$:    BNE     60$             ;
        BVC     50$             ;
        BIS     #6,(SP)         ; Set ZV
        BR      $CMPCS          ;
50$:    BIS     #4,(SP)         ; Set Z
        BR      $CMPCS          ;
60\:    BVC     $CMPCS          ;
        BIS     #2,(SP)         ; Set V

;
; **-\CMPCS-Completion routine which sets up only C in the PSW
;
; Copy only carry to stacked PSW.  Current stack:
;
;         (SP)    PSW with condition codes cleared
;        2(SP)    Completion routine address (to be discarded)
;        4(SP)    Return address
;

$CMPCS::ADC     (SP)            ; Set up carry
        MOV     4(SP),2(SP)     ; Set up return address for RTT
        MOV     (SP)+,2(SP)     ; And PSW. Returning to super mode?
70$:    BPL     80$             ; If PL yes
        MOV     #6,-(SP)        ; Number of bytes for (SP), PSW, and PC
        ADD     SP,(SP)         ; Compute clean stack value
        MTPI    SP              ; Set up previous stack pointer
80\:    RTT                     ; Return to previous mode and caller
        .DSABL  LSB
        .END
```

Example 9–5: Memory Allocation Map for TSUP

```
TSUP.TSK    Memory allocation map  TKB M43.00      Page 1
                    11-AUG-90   15:41

Partition name : GEN
Identification : 01
Task  UIC      : [30,55]
Stack    limits: 000274 001273 001000 00512.
PRG xfr address: 002130
Total address windows: 2.
Task  image  size  : 1344. words
Task address limits: 000000 005133
R-W disk blk limits: 000002 000007 000006 00006.

*** Root segment: TSUP

R/W mem  limits: 000000 005133 005134 02652.
Disk blk limits: 000002 000007 000006 00006.

Memory allocation synopsis:

Section                                        Title  Ident  File
-------                                        -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)  001274 002334 01244.
                           001274 001234 00668. TSUP   01    TSUP.OBJ
CMPAL :(RW,I,LCL,REL,CON)  000000 000474 00316.
PUR$D :(RO,I,LCL,REL,CON)  003630 000076 00062.
PUR$I :(RO,I,LCL,REL,CON)  003726 000752 00490.
$$RESL:(RO,I,LCL,REL,CON)  004700 000212 00138.
$$SLVC:(RO,I,LCL,REL,CON)  005112 000020 00016.

TSUP.TSK;1 Memory Allocation Map  TKB M43.00      Page 2
                    11-AUG-90   15:41

*** Task builder statistics:

    Total work file references: 2477.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 6988. words (27. pages)
    Size of work file: 1024. words (4. pages)

    Elapsed time:00:00:05
```

9.6.3.1 Building the Library SUPER

To build SUPER in directory [30,55] on device SY:, use the following TKB command sequence:

```
TKB> SUPER/-HD/LI/PI,SUPER/MA,SUPER=
TKB> LB:SYSLIB/LB:CMPAL,SY:[30,55]SUPER
TKB> /
Enter Options:
TKB> STACK=0
TKB> PAR=GEN:0:2000
TKB> CMPRT=$CMPCS
TKB> GBLXCL=$SAVAL
TKB> //
>
```

```
$ run $maksil
MAKSIL  V10.0-L
Resident Library name? SUPER
Task-built Resident Library input file <SUPER.TSK>?
Include symbol table (Yes/No) <Yes>?
Symbol table input file <SUPER.STB>?
Resident Library output file <SUPER.LIB>?
SUPER built in 1 K-words, 21 symbols in the directory
SUPER.TSK renamed to SUPER.TSK<40>
```

SUPER is built without a header or stack. It is position-independent and has only one program section, named .BLK. The /LI switch eliminates program section name conflicts between the library and the referencing task.

The completion routine module CMPAL is specified first in the input line. The library will run in partition GEN at 0 and is not more than 1K words.

The GBLXCL option excludes $SAVAL from the library's STB file. Exclude $SAVAL from the STB file because the referencing task, TSUP, also calls $SAVAL. If TSUP finds $SAVAL in the STB file of SUPER, it will not link a separate copy of $SAVAL into its task image from the system library. If TSUP could not link to a copy of $SAVAL that is mapped through user APRs, TSUP would call $SAVAL as a subroutine residing within the supervisor-mode library but without the necessary mode-switching vector and completion routine support. This option forces TKB to link $SAVAL from the system library into the task image for TSUP.

The memory allocation map in Example 9-2 shows the following information:

- SUPER begins at virtual 0.
- The completion routine, $CMPAL, is linked into the library from the system library at virtual 0.
- The entry point $CMPAL is located at virtual 22, SEARCH is located at 220, and SORT is located at 140. All of these entry points are relocatable.

9.6.3.2 Building TSUP

Use the following TKB command sequence to build a task, TSUP, that links to SUPER:

```
TKB>TSUP,TSUP=TSUP
TKB>/
Enter Options:
TKB>RESSUP=SUPER/SV:0
TKB>//
>
```

This command sequence tells TKB to include in the logical address space of TSUP a user-owned supervisor-mode library named SUPER. TKB includes a 4-word mode-switching vector within the task image for each call to a subroutine within the library. The library is position-independent and is mapped with supervisor I-space APR 0. This is a requirement for CSM libraries because the CSM library expects to find the entry point of the completion routine at location 10.

The memory allocation map for TSUP in Example 9-5 shows the following information:

- $CMPAL is linked from the STB file of the library and begins at location 0.
- The mode-switching vectors begin at 5112 and are 16 bytes in length. This means that TSUP calls subroutines within the library two times (four words for each vector).

Chapter 10

Task Builder Command Line Format

10.1 Running the Task Builder

To run the Task Builder, type:

```
RUN $TKB
```

Or, if the system manager has installed TKB as a concise command language (CCL) command, you can type:

```
TKB
```

The Task Builder responds with the prompt TKB> and you type a command. If TKB has been installed as a CCL command, you can type TKB and the command on the same line:

```
TKB command
```

10.1.1 Command Line

The Task Builder produces up to three files as output from its analysis of the object files you specify as input. The general form of the command is:

task-file,map-file,symbol-file=input-file,...,input-file

task-file — The file specification you give to the executable file produced by the Task Builder. If you do not want this file produced, type the comma delimiter. If you leave off the file type from the file specification, the Task Builder supplies a default type of .TSK.

map-file — The file specification you give to the memory map file produced by the Task Builder. If you do not want this file produced, type the comma delimiter. If you leave off the file type from the file specification, the Task Builder supplies a default type of .MAP.

symbol-file — The file specification you give to the symbol-table file produced by the Task Builder. If you do not want this file produced, simply leave out the file specification. If you leave off the file type from the file specification, the Task Builder supplies a default type of .STB.

input-files

The input to the Task Builder. For a simple (nonoverlaid) build, these are the object files produced from the assembly or compilation of your program and subroutines, plus disk library files containing subroutines needed to complete the program.

You signify disk library files by appending the switch /LB to the file specification. This notifies the Task Builder that the file named is a library to be searched. The Task Builder searches the library for any unresolved references in the object files appearing to the left of the library file in the command line.

If you do not specify file types, the Task Builder assumes a default type of .OBJ for object files and a default type of .OLB for object libraries.

For an overlaid build, the input file is an ODL file, signified with a /MP switch. The Task Builder assumes a default file type of .ODL for files with the /MP switch.

If you give a device designator or a project-programmer number in a file specification in the input list (to the right of the equal sign), they apply to all file specifications to the right in the list that do not have a device designator or a project-programmer number.

For a build using MACRO object programs, for example, a suitable command line is:

```
TKB EXE1,EXE1,EXE1=OBJ1,OBJ2,LB:RMSLIB/LB
```

The Task Builder constructs the executable file EXE1.TSK, the map file EXE1.MAP and the symbol table file EXE1.STB from the files OBJ1.OBJ, OBJ2.OBJ, and relevant modules from the library RMSLIB.OLB. (The relevant modules are those referenced in your program. You may have referred to them in source statements, or the MAC assembler may have translated source statements into calls referring to this library.)

To omit the map file, type:

```
TKB EXE1,,EXE1=OBJ1,OBJ2,LB:RMSLIB/LB
```

To produce only the executable file, type:

```
TKB EXE1=OBJ1,OBJ2,LB:RMSLIB/LB
```

To produce no output files, type:

```
TKB=OBJ1,OBJ2,LB:RMSLIB/LB
```

The example above is useful if you are running the Task Builder only to see error messages; that is, for a diagnostic run. Note how project-programmer numbers and device designators work when given for a file specification:

```
TKB=OBJ1,[2,243]OBJ2,OBJ3,LB:RMSLIB/LB,MYLIB/LB
```

For this command, the Task Builder would search for the file OBJ1.OBJ in the user's account. It would attempt to find the files OBJ2.OBJ and OBJ3.OBJ on the public disk structure in the account [2,243]. The project-programmer number also applies to the libraries. That is, the Task Builder would look on the system library disk for a file RMSLIB.OLB under the account [2,243]. Likewise, since the device name LB: also applies to MYLIB, the Task Builder would look on the system library disk in account [2,243] for the library file MYLIB.OLB.

If you do not want this to happen, you must respecify the project-programmer number and device that you want to apply to remaining files. The simplest way to accomplish this is to assign a logical name to the account [2,243] and use the system-wide logical SY: to "go back to" your account on the public disk structure. For example:

```
ASSIGN SY:[2,243] JOHN

Ready

TKB=JOHN:FILE1,SY:FILE2,FILE3,LB:RMSLIB/LB,SY:MYLIB/LB
```

This can also be accomplished using multiple command lines, as shown in the following section.

10.1.2 Multiline Command

Because you can specify any number of input files to the Task Builder, it is sometimes necessary to enter a command on more than one line.

If you run the Task Builder such that it prompts with TKB>, it continues prompting for input until it receives a line consisting only of two slash characters (//). For example:

```
RUN $TKB
TKB>IMG1,IMG1,IMG1=SY:[2,243]FILE1
TKB>FILE2,FILE3,LB:RMSLIB/LB
TKB>MYLIB/LB
TKB>//
```

This sequence produces the same result as the single line command:

```
TKB IMG1,IMG1,IMG1=JOHN:FILE1,SY:FILE2,FILE3,LB:RMSLIB/LB,SY:MYLIB/LB
```

In addition, it produces all three output files.

You must specify the output file specifications and the equal sign on the first command line. You can begin or continue input file specifications on subsequent lines.

10.2 Options

You may need to specify options to build a particular program. An option modifies the action taking place during the build. To include options, you must use the multiline format. If you specify a line consisting of a single slash (/), the Task Builder assumes that the last input file has been entered and prompts for options by displaying "ENTER OPTIONS:" and another TKB> prompt. You then enter the options you want and terminate the build with the double slash. For example:

```
RUN $TKB
TKB> command
TKB> continued-command
TKB> /
ENTER OPTIONS:
TKB> option
TKB> //
```

10.5 Comments in Lines

You can put comments anywhere in the command sequence. You begin a comment with a semicolon (;) and terminate it with a carriage return. For example, you could add comments to the indirect command file in the previous section as follows:

```
;
;BUILD 32T
;
;THE OUTPUT FILES ARE
;
IMG1,IMG2=
;
;THE INPUT FILES ARE
;
IN1,IN2,IN3
;
;
;OPTIONS ARE
;
/
COMMON=JRNAL:RO    ;RATE TABLES
;
//
```

10.6 File Specifications

You use the standard RSTS/E conventions for file specifications. In general, the format is:

device:[ppn]filename.type/sw1/sw2.../swn

If you do not specify a device, the public disk structure is assumed. The default for project-programmer number is your account. The exception is when you specify a device or project-programmer number for a file in a list of files. Such a device designator or project-programmer number "sticks" to all file specifications to the right. For example, consider the following input list:

```
TKB =OBJ1,[2,243]OBJ2,OBJ3,LB:RMSLIB/LB
```

The Task Builder looks for the file OBJ1.OBJ in the user's account. It looks for the files OBJ2.OBJ and OBJ3.OBJ on the public disk structure in account [2,243]. It looks for the file RMSLIB.OLB on the system library disk in account [2,243].

The default for file type depends on the switch you apply to the file specification. If no switches are used, the defaults are:

```
file.TSK,file.MAP,file.STB=file.OBJ,...,file.OBJ
```

Defaults assumed when you use various switches are described in Chapter 11. For example, the default file type when you use the /LB switch is .OLB.

Chapter 11

Task Builder Switches

The Task Builder lets you modify the action taken on a file by appending a switch to the file specification. A switch is a slash (/) followed by a two- to four-character code. In general, you can precede the two- to four-character code with a minus sign (-) or the letters "NO", and the Task Builder negates the function of the code. For example, the Task Builder recognizes the following settings for the switch /MP:

/MP The file is an ODL file.
/-MP The file is not an ODL file.
/NOMP The file is not an ODL file.

The Task Builder assumes a default setting for each switch. For example, if you do not specify any setting for the /MP switch, the Task Builder assumes /-MP (that the file is not an ODL file). In the switch descriptions in this chapter, note that the "Syntax" section shows where the switch is placed by using the opposite of the default setting. (There is no need to specify a switch if you want to use the default.)

Table 11–1 lists the Task Builder switches available on RSTS/E systems. They are described in following subsections in alphabetical order.

Table 11–1: Task Builder Switches

Switch	Meaning	Applies to File	Default
/CC	Input file consists of concatenated programs or subprograms.	.OBJ	/CC
/CO	Causes the Task Builder to build a shared common.	.TSK, .STB	/CO
/DA	Executable program contains a debugging aid.	.TSK, .OBJ	/-DA
/DL	Specified library file is a replacement for the default system library.	.OLB	/-DL
/EL	Extend library	.TSK	/-EL
/FM	Enables the Fast Map feature of the executive.	.MAP	/-FM
/FO	Indicates that the memory resident overlay should use the Fast Map version.	.MAP	/FO
/FP	Program uses Floating-Point processor.	.TSK	/FP

(continued on next page)

11.2 /CO—Build a Common Block Shared Region

File

Task image
.STB file

Syntax

file.TSK/CO=file.OBJ

or

,,file.STB/CO=file.OBJ

Description

The /CO switch informs the Task Builder that a shared common is being built. If you build a shared common, you should use the /CO switch and the /-HD switch.

If you use the /-PI switch for an absolute shared common, all the program sections in the common are marked absolute. Using the /-PI/-HD switches without the /CO switch causes the Task Builder to build a shared library.

If you use the /PI switch for a relocatable shared common, all program sections in the common are marked relocatable.

In either case, the .STB file contains all the program section names, attributes, lengths, and symbols. The Task Builder links common blocks by program sections. Therefore, the .STB file of a shared region built with the /CO switch contains all defined program sections.

Using the /PI/-HD switches without the /CO switch causes the Task Builder to build a shared common.

The /CO switch does not have a /-CO form.

Effect

This switch causes the Task Builder to include all program section declarations in the .STB file.

Defaults

/CO

Example

```
RUN $TKB
TKB>VAL/CO/-HD=VAL.OBJ
TKB>//
```

NOTE

Commons (read/write libraries) must still be processed using the MAKSIL utility. See the *RSTS/E Programmer's Utilities Manual* for more details about MAKSIL.

11.3 /DA—Debugging Aid

File

Executable program file or input file

Syntax

file.TSK/DA=file.OBJ

or

file.TSK=file.OBJ,file.OBJ/DA

Description

If you use the /DA switch on the executable program file, the Task Builder automatically includes the system debugging aid LB:ODT.OBJ in the executable program (LB:ODTID.OBJ if /ID is also included).

If you use this switch on one of your input files, the Task Builder assumes that the file is a debugging aid that you have written.

In either case, /DA has the following effects:

1. The transfer address of the debugging aid overrides the executable program transfer address.
2. The Task Builder initializes the header of the program so that, when your program is loaded, register R0 through R4 contain the following values:

   R0 Transfer address of program.

   R1 Task name in Radix-50 format (word 1). The Task Builder derives this name from the TASK=option. If no TASK= is supplied, this value will be 0.

   R2 Second word of task name.

   R3 The first three of six RAD50 characters representing the version number of your program. The Task Builder derives this number from the first .IDENT directive it encounters in your program. If no .IDENT directives appear, this value will be 0.

   R4 The second three RAD50 characters representing the version number of your program.

Refer to your specific language reference manual for more information about debugging aids.

Default

/-DA

Example

```
RUN $TKB
TKB>PROG/DA=OBJ,OBJ2,LB:F4POTS/LB
TKB>//
```

11.4 /DL—Default Library

File

Input

Syntax

file.TSK=file.OBJ,file.OLB/DL

Description

The library file you specify replaces the file LB:SYSLIB.OLB as the library file that the Task Builder searches to resolve undefined global references. This file is searched only when undefined symbols remain after all the files you specify have been processed. The /DL switch can be used with only one input file.

Default

/-DL

Example

```
RUN $TKB
TKB> PROG=PROG,LB:F4POTS/LB,NEWLIB/DL
TKB> //
```

11.5 /EL—Extend Library

File

Executable program file

Syntax

file.TSK/LI/-HD/EL=file.OBJ

Description

The /EL switch places the upper-address limit as determined by the PAR option in the library's label block, though the actual size of the library may be smaller. This switch is useful when building vectored libraries subject to size changes, such as RMS.

The switch specifies the maximum possible size for the library according to the size specified in the PAR option. The switch specifies a larger library virtual address range than is actually present in the library to allow RMS to map its vectored library segments.

Default

/-EL

11.6 /F ... g Point

ecutable program file

Syntax

file.TSK/FP=file.OBJ

Description

Setting the /FP switch causes the RSTS/E monitor to save the state of the floating-point processor when the program is run. You must set this switch on systems that have the floating-point processor (if the program uses the floating-point processor), so that the run-time system can trap floating-point errors properly. Setting or negating this switch has no effect on systems without a floating-point processor.

Default

/FP

Example

```
RUN $TKB
TKB>PROG/FP=OBJ1,OBJ2,LB:F4POTS/LB
TKB>//
```

11.9 /FU—Full Search

File

Executable program file

Syntax

file.TSK/FU=file.ODL/MP

Description

The /FU switch affects how the Task Builder inserts code from the default library when your overlay structure has co-trees. Normally, when the same code (program section) is called or referenced from different co-trees, it is built into both co-trees unless it can be resolved from code already built into the main root. This prevents the problem of run-time errors caused by unintentionally displacing segments with cross-tree calls, as described in Chapter 4.

If you use this switch, the Task Builder can resolve undefined global references with code from the default library that is already built into other co-trees. This can be useful if you want to try to cut down on the space taken by code inserted into co-trees from the default library, as described in Section 4.4.8.

Default

/-FU

Example

```
RUN $TKB
TKB>PROG/FU=OVERLY/MP
ENTER OPTIONS:
TKB>//
```

ader

xecutable program file or symbol definition file

Syntax

file.TSK/-HD,,file.STB=file.OBJ

or

file.TSK,,file.STB/-HD=file.OBJ

Description

The /HD switch causes the Task Builder to generate a header for your executable program file. This header is used by the run-time system when it loads your program for execution. The run-time system takes certain values from the header and inserts them in the low 1000 bytes of your program. (This area is used by the RSTS/E monitor, the run-time system, and—with a few languages—your program itself. For example, this area contains the "core common" area accessible to BASIC-PLUS-2 programs and the FIRQB and XRB areas used by MACRO programs. The contents of this area may be of interest to you if you are programming in MACRO. The area is described in the *RSTS/E System Directives Manual*.

In any case, you must have a header for executable program files (this is the default). If you are building a resident library or common, or a run-time system itself, you must negate this switch.

Default

/HD

Example

```
RUN $TKB
TKB> DATLIB/-HD/PI,,DATLIB/PI=DAT1.DAT,DAT2.DAT,DAT3.DAT
TKB> /
ENTER OPTIONS:
TKB> PAR=DATLIB
TKB> //
```

11.11 /ID—I- and D-Space

File

Task image (.TSK).

Syntax

file.TSK/ID=file.OBJ,file.OLB

Description

Use this switch to create I- and D-space tasks. The switch directs the Task Builder to mark your task as one that uses I-space APRs and D-space APRs in user mode. The Task Builder separates I-PSECTs from D-PSECTs. See Chapter 8 for more information.

Default

/-ID

Example

```
RUN $TKB
TKB> PROG1.TSK/ID=PROG1.OBJ,PROG1.OLB/LB
TKB> //
```

11.13 /LI—Build a Library Shared Region

File

Task Image
.STB file

Syntax

file.TSK/LI[:apr bit mask]

or

,,file.STB/LI[:apr bit mask]

Description

The /LI switch makes the Task Builder build a shared library. However, you must use the /-HD switch with the /LI switch to build the shared library. The /LI switch does not have a /-LI form.

See the discussion on APR masks in Chapter 7.

Effect

The Task Builder includes only one program section declaration in the .STB file.

If you use the /-PI switch for an absolute library, the Task Builder names the program section .ABS, makes the library position dependent, and defines all symbols as absolute. Also, if you use the /-PI switch without the /LI switch, the Task Builder assumes /LI to be the default.

If you use the /PI switch for a relocatable library, the Task Builder names the program section the same as the root segment of the library. The Task Builder forces this name to be the first and only declared program section in the library. The Task Builder declares all global symbols in the .STB file relative to that program section. Also, if you use the /PI switch without the /LI switch, the Task Builder assumes that a shared common is to be built. (/CO is the default.)

Default

/LI [:nnn]

where:

n is the bit mask representing the APR in I-space used by the library

Example

```
RUN $TKB
TKB>PARODI/LI/-HD=PARODI.OBJ
TKB>//
```

NOTE

Libraries must still be processed using the MAKSIL utility. See the *RSTS/E Programmer's Utilities Manual* for more details about MAKSIL.

11.14 /MA—Map Contents of File

File

Input or memory allocation (map) file

Syntax

file.TSK,file.MAP=file.OBJ,file.OBJ/-MA

or

file.TSK,file.MAP/MA=file.OBJ

Description

If you negate this switch and apply it to an input file, the Task Builder leaves the file off the "file contents" portion of the memory map. Furthermore, it will exclude from the map all global symbols defined or referred to in the file.

If you set this switch for the map file, the Task Builder includes in the map the names of routines it has added to your program from the default library (LB:SYSLIB.OLB). It also includes in the map file information contained in the symbol definition file of any shared region referred to by the program.

Default

/MA for input files

/-MA for system library and resident library .STB files.

/-MA for map file

Example

```
RUN $TKB
TKB>PROG,PROG/MA=OBJ1,OBJ2,LB:F4POTS/LB
TKB>//
```

11.15 /MP—Overlay Map

File

Input

Syntax

file.TSK=file.ODL/MP

Description

Your input file is an overlay map (ODL file). The file contains directions for an overlay structure in the Overlay Description Language. When you use the switch, it must be the only input file that you specify. The default file type for a file with the /MP switch is .ODL.

Default

/-MP

Example

```
RUN $TKB
TKB>PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB>//
```

NOTE

If you use the multiline command format when you specify an ODL file, TKB automatically prompts for option input. Therefore, you must not use the single slash (/) to direct TKB to switch to option input mode when you have specified /MP on your input file.

11.16 /MU—Multiuser Program

File

Executable program file

Syntax

file.TSK/MU=file.OBJ

Description

The /MU switch tells the Task Builder to separate the program's read-only and read/write program sections. On RSTS/E systems, you only use this switch if you want to build a program so that the read-only code from the *root* is accessible to multiple users. For this reason, it is recommended that programs built with the /MU switch be nonoverlaid. Several steps are involved in this procedure.

When you use /MU on the executable file, the Task Builder places the read-only sections in your program's upper virtual address space and the read/write program sections in your program's lower virtual address space. You then have to use the MAKSIL program (described in the *RSTS/E Programmer's Utilities Manual* to make the read-only code accessible to multiple users, and add the read-only code as a resident area using the DCL INSTALL/LIBRARY command (described in the *RSTS/E System Manager's Guide*).

Multiple users can then run the program built, causing multiple copies of the read/write code to be executed, but with only one copy of the read-only code taking space in memory.

Note that a program built with the /MU switch cannot be run correctly until it has been converted by MAKSIL into a separate executable file (consisting of the read/write code) and a resident area, which in turn must be added with the DCL INSTALL/LIBRARY command (just like any resident area). Note also that, when you build a program and use the /MU switch, you can also use the HISEG option. If you build with HISEG, the Task Builder will put the read-only code to occupy virtual address space below the run-time system. If you do not build with HISEG, the Task Builder will put the read-only code so that it occupies the highest possible address space (using APR 7).

Default

/-MU

Example

```
 RUN $TKB
TKB> PROG/MU=OBJ1,OBJ2,OBJ3
TKB> //
```

11.21 /SB—Slow Build

File

Executable program file

Syntax

file.TSK/SB=file.obj

Description

The /SB qualifier causes the task to be built with the slow mode of the Task Builder. This option increases the amount on TKB internal storage and therefore allows you to build larger or more complex tasks.

Default

/-SB

Example

```
RUN $TKB
TKB>PROG/SB=OVERLY/MP
ENTER OPTIONS:
TKB//
```

11.22 /SG—Segregate Program Sections

File

Task image

Syntax

file.TSK/SG=file.OBJ

Description

The /SG switch allocates virtual address space to all read/write (RW) program sections and then to all read-only (RO) program sections.

Effect

The /SG switch gives you control over the ordering of program sections. By using the /SG switch, you cause the Task Builder to order program sections alphabetically by name within access code (RW followed by RO). If you specify the /SQ switch with the /SG switch, the Task Builder orders program sections in their input order by access code. (See the description of the /SQ switch for more information.)

You use the negated switch, /-SG, to make the Task Builder interleave the RW and RO program sections. Thus, the combination /-SG/SQ results in a task with its program sections allocated in input order and its RW and RO sections interleaved. Also, you can use /-SQ/-SG to make the Task Builder order program sections alphabetically with RW and RO sections interleaved. However, /SG is the default.

When task building multiuser tasks, the /MU switch causes the Task Builder to default to /SG. Therefore, to correctly build read-only tasks, you can use the /MU switch only.

Default

/SG

Example

```
 RUN $TKB
TKB> BARBEL/SG=BARBEL.OBJ
TKB> //
```

11.23 /SH—Short Map

File

Memory allocation (map) file

Syntax

file.TSK,file.MAP/-SH=file.OBJ

Description

Negating this switch (-SH) requests the long version of the memory allocation map. The Task Builder produces the "file contents" section of the map. An example of the long version of the map is shown in Example 11–1. The letters in brackets and the numbers in cirlces in the figure correspond to the notes following the figure.

Default

/SH

Example

```
 RUN $TKB
TKB> PROG,PROG/-SH=OBJ1,OBJ2,LB:F4POTS/LB
TKB> //
```

Example 11–1: Memory Allocation (Map) File

```
ROOTM.TSK   Memory allocation map  TKB 08.006      Page 1
                    05-MAY-90   13:50

Task    name   : ROOTM   [A]

Partition name : GEN   [B]
Identification : 01   [C]
Task  UIC      : [2,234]   [D]
Task  priority : 50.   [E]
Stack    limits: 001000 001777 001000 00512.   [F]
ODT xfr address: 011054   [G]
PRG xfr address: 002000   [H]
Task attributes: DA,MU   [I]
Total address windows: 2.   [J]
Task   extension   : 128. words   [K]
Task  image  size  : 9760. words   [L]
Total  task  size  : 9888. Words   [M]
Task address limits: 000000 046033   [N]
R-W disk blk limits: 000002 000101 000100 00064.   [O]
R-O disk blk limits: 000102 000112 000011 00009.   [P]

ROOTM.TSK Overlay description:

Base    Top         Length
----    ---         ------
000000  016027  016030  07192.       ROOTM
016030  031247  013220  05776.            MULOV
016030  032043  014014  06156.            ADDOV
032044  046033  013770  06136.                 SUBOV
032044  045667  013624  06036.                 DIVOV

ROOTM.TSK   Memory allocation map  TKB 08.006      Page 2
ROOTM                 05-MAY-90   13:50

*** Root segment: ROOTM   [A]

R/W mem  limits: 000000 016027 016030 07192.   [B]
R-O mem  limits: 160000 170577 010600 04480.   [C]
Disk blk limits: 000002 000020 000017 00015.   [D]

Memory allocation synopsis:

Section                                          Title Ident  File
-------                                          ----- -----  ----
. BLK.:(RW,I,LCL,REL,CON)      002000 000000 00000.   [E]
CODE  :(RW,I,LCL,REL,CON)      002000 000024 00020.
                               002000 000024 00020. .MAIN.01 ROOTM.OBJ  [F]

                        .
                        .
                        .
Global symbols:

ADD 170076-R  DATEND  010010-R DAT1  002024-R MUL  170066-R .ODTL1 010434-R  [G]
BEG 002000-R  DAT0    160000-R  DIV  170116-R SUB  170106-R .ODTL2 010436-R

File: ROOTM.OBJ  Title: .MAIN.  Ident: 01   [H]
    <. ABS.>: 000000 000000 000000 00000.   [I]
 >>>>>>>>>>>> Undefined reference: NOSYMB   [J]
```

(continued on next page)

Example 11–1 (Cont.): Memory Allocation (Map) File

```
    <CODE  >: 002000 002023 000024 00020.
       BEG    002000-R    [K]
    <DATA  >: 160000 170041 010042 04130.    [L]
                          .
                          .
                          .
************

Undefined references:    [M]

    NOSYMB

ROOTM.TSK   Memory allocation map  TKB 08.006      Page 4
MULOV                 05-MAY-90   13:50

*** Segment: MULOV

R/W mem  limits: 016030 031247 013220 05776.
Disk blk limits: 000021 000034 000014 00012.

Memory allocation synopsis:

Section                                            Title  Ident  File
-------                                            -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)      016030 013220 05776.
                               016030 013220 05776. .MAIN.       MULOV.OBJ
$$ALVC:(RO,I,LCL,REL,CON)      031250 000000 00000.
$$RTS :(RO,I,GBL,REL,OVR)      170570 000002 00002.

Global symbols:

MUL    016030-R

File: MULOV.OBJ  Title: .MAIN.  Ident:
    <. BLK.>: 016030 031247 013220 05776.
       MUL    016030-R
                          .
                          .
                          .

*** Task builder statistics:

    Total work file references: 4693.    [A]
    Work  file  reads: 0.    [B]
    Work  file writes: 0.    [B]
    Size of core pool: 4814. words (18. pages)    [C]
    Size of work file: 3072. words (12. pages)    [D]

    Elapsed time:00:00:17    [E]
```

❶ The page header shows the name of the executable program file and the overlay segment name (if applicable), along with the date, time, and version of the Task Builder that created the map.

❷ The task attribute section contains the following information:

A. Task Name. The name specified in the TASK option. If you do not use the TASK option, the Task Builder suppresses this field.

B. Partition Name. The partition specified in the PAR option. If you do not specify a partition, the default is a partition named GEN.

C. Identification. The version as specified in the .IDENT assembler directive. If you do not specify, the default is the same as the version of the Task Builder.

D. User Identification Code. The project-programmer number used to create the executable program file.

E. Priority. (On RSTS/E systems, this field is ignored.) Priority is suppressed if you do not use the PRI= option.

F. Stack Limits. The low and high octal addresses of the stack, followed by its length in octal and decimal bytes.

G. ODT Transfer Address. The starting address of the ODT debugging aid. If you do not specify the ODT debugging aid, this field is suppressed.

H. Program Transfer Address. The starting address of your program. For MACRO programmers, this is the address of the symbol specified in the .END directive of the source code of your program. (The compilers generate a starting address automatically.) If you do not specify a transfer address for your program, the Task Builder automatically establishes a transfer address of 000001 for it. The Task Builder also suppresses this field in the map if no transfer address is specified.

I. Attributes. Using certain switches indicates that your program has certain attributes. Such switch settings are shown only if they differ from the defaults. For example, the following could be displayed:

DA - the program contains a debugging aid.

MU - the program is broken into RO and RW sections for processing by MAKSIL. See the description of the /MU switch in Section 11.16 of this manual, and the MAKSIL chapter in the *RSTS/E Programmer's Utilities Manual*, for more information.

J. Total Address Windows. The number of window blocks allocated to the program.

K. Task Extension. The increment of physical memory (in decimal words) allocated through the EXTTSK or PAR option.

L. Task Image Size. The amount of memory (in decimal words) required to contain your program's code. This number does not include physical memory allocated through the EXTTSK option.

M. Total Task Size. The amount of memory (in decimal words) allocated to your program, including the physical memory allocated through the EXTTSK option or PAR option.

N. Task Address Limits. The lowest and highest virtual addresses allocated to the program, exclusive of resident areas.

O. Read/Write Disk Block Limits. From left to right: the first octal relative disk block of the program's read/write region; the last octal relative disk block number of the read/write region; the total contiguous disk blocks required to accomodate the read/write region in octal and decimal.

P. Read-Only Disk Block Limits. From left to right: the first octal relative disk block of the multiuser program's read-only region; the last octal relative disk block number of the read-only region; the total contiguous disk blocks required to accomodate the read-only region in octal and decimal. This field appears only when you are building a multiuser program with the /MU switch.

❸ The Overlay Description shows, for each overlay segment in the tree structure of an overlaid program, the beginning virtual address (the base), the highest virtual address (the top), the length of the segment in octal and decimal bytes, and the segment name. Indenting is used to illustrate the ascending levels in

11.24 /SP—Spool Map Output

File

Memory allocation (map) file

Syntax

file.TSK,file.MAP/SP=file.OBJ

Description

This switch determines whether your map file is automatically queued to the line printer for output. If you use this switch, the Task Builder creates a map file and queues it for printing. The default (if you specify a map file) is to create the map file but not to queue it for printing.

Default

/-SP

Example

```
RUN $TKB
TKB>PROG,PROG/SP=OBJ1,OBJ2,LB:F4POTS/LB
TKB>//
```

11.25 /SQ—Sequential

File

Executable program file

Syntax

file.TSK/SQ=file.OBJ

Description

If you set the /SQ switch, the Task Builder does not reorder program sections alphabetically. Instead, it collects all the references to a given program section from your input files, groups them according to their access code (read-only or read/write), and within these groups, allocates memory for them in the order that you input them.

You use this switch to satisfy requirements that certain program sections be adjacent. Using this feature is otherwise discouraged because standard library routines (such as FORTRAN I/O handling routines and File Control System (FCS) routines from SYSLIB) will not work properly.

You can also make program sections adjacent by selecting their names alphabetically to correspond to the desired order.

Default

/-SQ

Example

```
RUN $TKB
TKB>PROG/SQ=OBJ1,OBJ2,OBJ3
TKB>//
```

11.26 /SS—Selective Search

File

Input file

Syntax

file.TSK=file.OBJ/SS

or

file.TSK=file.OBJ,file.STB/SS

or

file.TSK=file.OBJ,file.OLB/LB/SS

Description

Setting the /SS switch tells the Task Builder to include in its internal symbol table only those global symbols for which it has already encountered an undefined reference.

When processing an input file, the Task Builder normally includes into its internal symbol table each global symbol it encounters within the file whether or not there are references to it. When you attach the /SS switch to an input file, the Task Builder checks each global symbol it encounters within that file against its list of undefined references. If the Task Builder finds a match, it includes the symbol into its symbol table.

Default

/-SS

Example

Suppose that you are building a program consisting of input files containing global entry points and references (calls) to them, as shown in Table 11–2.

Table 11–2: Input Files for /SS Example

Input File Name	Global Definition	Global Reference
IN1.OBJ		A
IN2.OBJ	A B C	
IN3.OBJ		C
IN4.OBJ	A B C	

Files IN2 and IN4 contain definitions for global symbols of the same name. Assume that the global symbols represent entry points to different routines within these files.

Suppose that you want the Task Builder to resolve the reference to A in IN1 with the definition of A in IN2. Further, assume that you want the reference to global symbol C in IN3 to be resolved with the definition of C in IN4. You can accomplish this by ordering the input files and using the /SS switch. For example:

```
TKB>SELECT=IN1,IN2/SS,IN3,IN4/SS
```

The Task Builder processes input files from left to right. Thus, the Task Builder processes file IN1 first and finds the reference to symbol A. Since there is no definition for A within IN1, the Task Builder marks A as undefined and moves on to process IN2. Because IN2 has the /SS switch, the Task Builder limits its search of IN2 to symbols it has already marked as undefined, namely A. The Task Builder finds a definition for A and puts A in its symbol table.

The Task Builder moves on to IN3, and encounters the reference to symbol C. Since the Task Builder did not include symbol C from IN2 in its symbol table, it marks C as undefined and moves on to IN4. When the Task Builder processes IN4, it finds the definition for C, and includes that symbol in the table. Again, since the /SS switch is attached, only symbol C is included in the Task Builder's internal symbol table.

Thus, the reference to A in IN1 is resolved with the definition in IN2, and the reference to C in IN3 is resolved with the definition in IN4. Note that the /SS switch affects only the Task Builder's internal symbol table. The routines for which symbols B and C are entry points will be included in the executable program file even though there are no references to them.

11.27 /TR—Traceable Program

File

Executable program file

Syntax

file.TSK/TR=file.OBJ

Description

When this switch is set, the Task Builder sets the T-bit in the initial program status word (PSW) for your program. When your program is executed, a trace trap occurs when each instruction is completed.

The system library (SYSLIB.OLB) contains a trace routine (TRACE.OBJ) that processes the trap. You must explicitly build this routine into your executable file if you want to use it. To do this, you must use the LBR utility (see the *RSTS/E Programmer's Utilities Manual*) to remove TRACE.OBJ from the system library. You then build TRACE.OBJ into your program using the /DA switch. The example below shows TRACE.OBJ in the user's account on the public structure.

Default

/-TR

Example

```
RUN $TKB
TKB>PROG/TR=OBJ1,OBJ2,OBJ3,TRACE/DA
TKB>//
```

11.28 /WI—Wide Listing Format

File

Memory allocation (map) file

Syntax

file.TSK,file.MAP/-WI=file.OBJ

Description

Negating this switch causes the Task Builder to format the map file 80 columns wide. Setting the switch or accepting the default causes a map 132 columns wide.

Default

/WI

Example

```
 RUN $TKB
TKB> PROG,PROG/-WI=OBJ1,OBJ2,LB:F4POTS/LB
TKB> //
```

Table 12–1 (Cont.): Task Builder Options

Option	Meaning
RESLIB	Declares a user-owned resident library to be accessed by the program.
RESSUP	Declares that your task intends to access a user-owned supervisor-mode library.
RNDSEG	Rounds the size of a named segment up to the nearest APR boundary while building a resident library.
STACK	Defines the size of the stack.
SUPLIB	Declares that your task intends to access a systemwide supervisor-mode library.
TASK	Names the executable program for SYSTAT.
TSKV	Declares the address of the program's SST vector.
UNITS	Declares the maximum number of units (channels).
VARRAY	Specifies an overlaid virtual array so that each segment of an overlaid task that uses it defines the array in the same way that it is defined in the root segment.
VSECT	Specifies the virtual base address, virtual length, and physical memory allocated to the named program section.
WNDWS	Declares the number of additional address windows to be used by the program.

12.1 ABORT—Abort the Build

The ABORT option is useful when you have made an error on an earlier line of Task Builder input. When you type the ABORT=n in response to a TKB> option prompt, the Task Builder stops accepting input for the current build and prepares to accept input for a new build operation. You can then restart the same or another command sequence.

Syntax

ABORT=n

where:

n is any integer. (You must specify =n to satisfy the general form of the syntax for options, but the value is ignored.)

Note that typing a Ctrl/Z (pressing the Ctrl and Z keys at the same time) causes the Task Builder to stop accepting input and start building the current program. ABORT is the only way to restart the Task Builder if you find an error and do not want a build to take place.

Default

None

Example

```
 RUN $TKB
TKB> PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB> RESLIB=RMSRES
TKB> ABORT=1
?TKB -- *FATAL* -- TASK BUILD ABORTED VIA REQUEST

ABORT = 1

TKB>
```

access-code

is either RW (read/write) or RO (read-only). This code indicates how your program intends to access the library. (It will be RO for Digital-provided resident libraries such as BP2RES, FDVRES, C81CIS, etc.) For example:

```
TKB>CLSTR=C81CIS,FDVRES:RO
```

apr

is an integer in the range of 1 to 7 that specifies the first Active Page Register (APR) reserved for the clustered libraries. (See Section 2.3.4 for information on APRs.) If you leave this parameter off, the Task Builder assigns the highest APRs it can to the cluster (APRs 6 and 7 for the above command line).

Currently, Digital-supplied libraries are built to use APRs 6 and 7 so that they can occupy 8K words at the highest end of the user job area.

Default

None

Example

```
RUN $TKB
TKB>PROG,PROG=PROG,LB:C81CIS/LB
TKB>/
ENTER OPTIONS:
TKB>CLSTR=C81CIS,FDVRES:RO
TKB>//
```

12.6 CMPRT—Completion Routine

Use this option to identify a shared region as a supervisor mode library. The CMPRT option requires an argument that specifies the entry point of the completion routine in the library. The completion routine switches the processor back from supervisor mode to user mode, and returns program control to the user task after the supervisor mode library subroutine finishes executing. The completion routine is invoked by a RTS PC at the end of the supervisor mode subroutine.

The following completion routines are available in the system library:

- $CMPCS restores only the carry bit in the user mode PSW.
- $CMPAL restores all the condition code bits in the user mode PSW.

These routines perform the necessary overhead to switch the processor from supervisor mode to user mode and return program control to the user task at the instruction following the call to supervisor mode library subroutine. Although you can write your own completion routines, you should use either $CMPCS or $CMPAL whenever possible. Chapter 9 discusses these routines in detail.

Syntax

CMPRT=name

where:

name	is a one- to six-character name identifying the completion to routine global entry point.

Default

None

12.7 COMMON—Access System Common Block

The COMMON option indicates a resident library that should contain only data. The format of the COMMON option is the same as the LIBR option (Section 12.18).

Syntax

COMMON=name:access-code[:apr[:mask]]

See the description of the LIBR option (Section 12.18) for a discussion of the parameters.

Example

```
RUN $TKB
TKB>PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB>COMMON=MYCOM:RW:5
TKB>//
```

12.8 DSPPAT—Absolute Patch for D-Space

Use the DSPPAT option to declare a series of object-level patches starting at the specified base address, and to make patches to the D-space and I- and D-space task. A maximum of eight patches can be specified. You can also use this option to patch a conventional task at any location.

Syntax

DSPPAT=seg-name:address:val1:val2:...:val8

where:

seg-name	is the 1- to 6-character Radix-50 name of the segment.
address	is the octal address of the first patch. The address can only be on a byte boundary; however, two bytes are always modified for each patch: the addressed byte and the following byte.
val1	is an octal number in the range of 0 to 177777 to be stored at address.
val2	is an octal number in the range of 0 to 177777 to be stored at address+2.
.	
.	
.	
val8	is an octal number in the range of 177777 to be stored at address+16.

Default

None

Example

```
RUN $TKB
TKB>PROG,PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB>
ENTER OPTIONS:
TKB>DSPPAT:MYDATA:100:1:2
```

The DSPPAT option sets the word at location 100 in segment MYDATA to 1, and the word at location 102 in segment MYDATA to 2.

NOTE

All patches must be within the segment address limits or TKB generates the following error message:

```
TKB--*DIAG*--Load address out of range in module-name
```

12.9 EXTSCT—Extend Program Section

The EXTSCT option extends the size of a program section. If the program section has the concatenated (CON) attribute, its size is extended by the length specified. If the program section has the overlay (OVR) attribute, its size is set equal to the length specified, if the length specified is greater than the current size. (If the length is less than the current size, the current size is allocated.)

Syntax

EXTSCT=psect-name:length

where:

psect-name	is the one- to six-character name of the program section to be extended.
length	is the octal number of bytes to extend the program section.

Default

None

Example

```
RUN $TKB
TKB> PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> EXTSCT=BUFF:250
TKB> //
```

Suppose that BUFF is initially 200[8] bytes long. After the above option is specified, it will be allocated 450[8] bytes if it is concatenated (CON), or 250[8] bytes if it is overlaid (OVR).

12.10 EXTTSK—Extend Task Memory

The EXTTSK option directs the system to allocate additional memory for your executable program, up to a maximum of (32K-32736) words.

The amount of memory available to the program is the sum of the program's size plus the increment you specify in the EXTTSK option, rounded up to the next 32-word boundary. This option extends only the D-space of an I- and D-space task.

This option extends only the D-space of and I- and D-space task.

Syntax

EXTTSK=length

where:

length — is a decimal number in the range 0<n<32,736 specifying the increase in task memory allocation (in words).

Default

The program is extended to the next multiple of 1K words.

Example

```
 RUN $TKB
TKB> PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> EXTTSK=4096
TKB> //
```

NOTE

1. You should not use the EXTTSK option to extend a task containing memory-resident overlays because the system does not map the extended area.

2. Be careful when extending an I- and D-space task that is linked to a library containing both data and instructions. Normally, libraries are mapped in both I- and D-space, allowing data and instructions to be intermixed. The extension length must not extend into the area mapped for the library or the library will be mapped in I-space only.

12.11 FMTBUF—Format Buffer Size

The FMTBUF option declares the length of the internal working storage that you want the Task Builder to allocate within your program for the compilation of format specifications at run time. The length of this area must equal or exceed the number of bytes in the longest format string to be processed.

Run-time compilation occurs whenever an array is referred to as the source of formatting information within a FORTRAN I/O statement. The program section that the Task Builder extends has the reserved name $$OBF1.

Syntax

FMTBUF=n

where:

n is a decimal integer, larger than the default (132), that specifies the number of characters in the longest format specification.

Default

FMTBUF=132

Example

```
RUN $TKB
TKB>PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB>/
ENTER OPTIONS:
TKB>FMTBUF=140
TKB>//
```

12.12 GBLDEF—Define a Global Symbol

The GBLDEF option defines a global symbol and its value. The Task Builder considers this symbol definition to be absolute. It overrides any definition in your object program files.

Syntax

GBLDEF=symbol-name:symbol-value

where:

symbol-name	is the one- to six-character name assigned to the global symbol.
symbol-value	is an octal number in the range of 0 through 177777 assigned to the defined symbol.

Default

None

Example

```
RUN $TKB
TKB>PROG=OBJ1,OBJ2,F4POTS/LB
TKB>/
ENTER OPTIONS:
TKB>GBLDEF=LITVAL=1357
TKB>//
```

12.13 GBLINC—Include Global in .STB File

The GBLINC option includes a global symbol in the .STB file that would not otherwise be there. This option is used in Digital-supplied resident libraries that may need to call routines in other resident libraries in a cluster. It is also useful if you are building your own clusterable resident libraries.

Syntax

GBLINC=symbol

where:

symbol — is the global symbol name to be included in the symbol table file being built for the resident library.

Default

None

Example

```
 RUN $TKB
TKB> PROG,PROG,PROG=OVERLY/MP
TKB> /
Enter Options:
TKB> GBLINC=.FCSJT
TKB> GBLINC=USER
TKB> //
```

The GBLINC option includes the symbols named (.FCSJT and USER) in the symbol table file (PROG).

12.14 GBLPAT—Global Relative Patch

The GBLPAT option declares a series of object-level patch values starting at an offset relative to a global symbol. You can specify up to eight patch values.

Note that all patches must be within the segment address limits or the Task Builder will generate a fatal error.

Syntax

GBLPAT=seg-name:sym-name[+/-offset]:val1:val2:...:val8

where:

seg-name	is the one- to six-character name of the segment.
sym-name	is the one- to six-character name specifying the global symbol.
offset	is an octal number specifying the offset from the global symbol.
val1	is an octal number in the range of 0 through 177777 to be stored at the address of the global symbol plus or minus the offset.
val2	is an octal number in the range of 0 through 177777 to be stored at the address of the global symbol, plus or minus the offset, plus 2.
.	
.	
.	
val8	is an octal number in the range of 0 through 177777 to be stored at the address of the global symbol, plus or minus the offset, plus 14.

Default

None

Example

```
 RUN $TKB
TKB> PROG,PROG=OVERLY/MP
TKB> /
ENTER OPTIONS:
TKB> GBLPAT=IN1:MRTN+4:10001
TKB> //
```

The GBLPAT option sets the word at location MRTN+4 in segment IN1 to 010001.

12.15 GBLREF—Global Symbol Reference

The GBLREF option declares a global symbol reference. The reference originates in the root segment of the executable program.

Syntax

GBLREF=symbol-name

where:

symbol-name is the one- to six-character name of a global symbol.

Default

None

Example

```
RUN $TKB
TKB> PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB> GBLREF=MRTN
TKB> //
```

12.16 GBLXCL—Exclude Global from .STB File

The GBLXCL option excludes a global symbol from the .STB file that would otherwise be there. This option is used in Digital-supplied resident libraries that may need to call routines in other resident libraries in a cluster. It is also useful if you are building your own clusterable resident libraries.

Syntax

GBLXCL=symbol

where:

symbol — is the global symbol name to be excluded from the symbol table file being built for the resident library.

Default

None

Example

```
 RUN $TKB
TKB> PROG,PROG,PROG=OVERLY/MP
TKB> /
Enter Options:
TKB> GBLXCL=.FCSJT
TKB> GBLXCL=USER
TKB> //
```

The GBLXCL option excludes the symbols named (.FCSJT and USER) from the symbol table file (PROG).

12.17 HISEG—Define High Segment

The HISEG option associates an executable program with a user-written high segment, or run-time system. If there are global definitions within the high segment that resolve references in the input files you specify, the Task Builder links them correctly. The symbol-table file (.STB file) for the named run-time system must be in the account specified by the system logical name LB:. If the HISEG option is not specified:

1. The run-time system associated with the executable program is the same as that associated with the Task Builder itself.
2. No global references to symbols in that high segment are resolved.

Note that the HISEG option is sometimes used when you build a multiuser program with the /MU switch. (See Section 11.16.)

Syntax

HISEG=high-segment-name

where:

high-segment-name	is a one- to six-character name specifying the run-time system.

Default

If no high segment is specified, the run-time system associated with the Task Builder is assumed.

Example

```
 RUN $TKB
TKB> PROG=OBJ1,OBJ2
TKB> /
ENTER OPTIONS:
TKB> HISEG=USRTS
TKB> //
```

12.18 LIBR—Access System-Owned Resident Library

The LIBR option declares that your program intends to access a system-owned resident library.

Syntax

LIBR=name:access-code[:apr[:mask]]

where:

name — is the one- to six-character name specifying the library. The Task Builder expects to find a symbol table file and task image file of the same name (name.STB and name.TSK) on the device and under the account specified by the system logical name LB:.

The easiest way to find out if the files exist on LB: is to do a directory:

```
DIR LB:RMSRES.STB
  Name Typ  Size   Prot        DR3:[1,11]
RMSRES.STB     4  < 40>
DIR LB:RMSRES.TSK
  Name Typ  Size   Prot        DR3:[1,11]
RMSRES.TSK    16  < 40>
```

(If the files do not exist on LB:, you must use the RESLIB option, Section 12.23.)

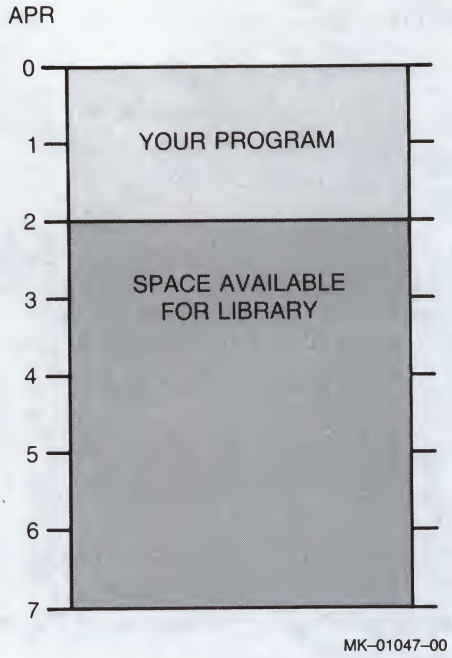
access-code — is the code RW (for read/write) or RO (for read-only), indicating the type of access required by your program.

mask — can be an explicit D-APR usage mask for the library. The value given here will override what was defined for the mask when you built the library, and will be acceptable to all other APR masks associated with this task. See Section 7.7 for more details.

apr — is an integer in the range of 1 to 7 that specifies the first Active Page Register (APR) reserved for the library.

It is not really necessary to understand Active Page Registers to use this modifier. Think of your 32K-word user job area as divided into 8 parts of 4K words each, numbered from 0 through 7. Your program occupies one or more of the lowest-numbered segments.

You can "map" a resident library into the area between the top of the task and the top of the virtual address space. The map must begin on a 4K-word boundary. For example, if the program takes 6K words, there are six APRs available (24K words) for resident library mapping. You can map up to 20K words of resident library into your job, beginning with APR 2, as illustrated in the following diagram:



Default

None

Example

```
RUN $TKB
TKB> PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB> LIBR=RMSRES:RO:5
TKB> //
```

This example causes the RMSRES library in LB: to be mapped through APRs 5 and 6. The run-time system is to be mapped through APRs 4 through 7.

12.19 MAXBUF—Maximum Record Buffer Size

The MAXBUF option declares the maximum record buffer size required for any file used by the program. If your program requires a maximum record size that exceeds the default buffer length (133 bytes), you must use this option to extend the buffer.

You must also include a language library (object time system, or OTS), such as FORTRAN's F4POTS, in your executable program for the extension to take place. The program section that is extended has the reserved name $$IOB1.

Syntax

MAXBUF=n

where:

n is a decimal integer, larger than 133, that specifies the maximum record size in bytes.

Default

MAXBUF=133

Example

```
 RUN $TKB
TKB> PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> MAXBUF=166
TKB> //
```

12.20 ODTV—ODT SST Vector

The ODTV option declares that a global symbol is the address of the ODT Synchronous System Trap vector table. You must define the global symbol in the main root segment of your program.

The vector address list contained at the global symbol will automatically be installed as the task is loaded at execution time. Also, the vectors are active on the execution of the first instruction when the task is started. This means that there is no window where the traps could be missed, and the program will not waste code space making explicit or SVTK$ or SVDB$ calls.

Syntax

ODTV=symbol-name:vector-length

where:

symbol-name	is a one- to six-character name of a global symbol.
vector-length	is a decimal integer in the range of 1 through 32, specifying the length of the SST vector in words.

Default

None

Example

```
 RUN $TKB
TKB> PROG/DA=OBJ1,OBJ2
TKB> /
ENTER OPTIONS:
TKB> ODTV=TRPVEC:8
TKB> //
```

For related information, refer to the *RSTS/E System Directives Manual* for the SVDB$ (Set SST Vector Table for Debugging Aid) macro.

12.21 PAR—Partition for Resident Area

The PAR option is used when building a resident area. The option identifies a "partition" for the resident area: the amount of space the resident area will occupy when linked into user programs in the user job area (and its location, if the resident library is to occupy absolute addresses).

Syntax

PAR=pname[:base:length]

where:

pname — is the name of the partition. This name must be the same as the file name portion of the executable and symbol table files in the command line. For example:

```
RUN $TKB
TKB>LIBRES/-HD,,LIBRES/PI=LIBRES
TKB>/
ENTER OPTIONS:
TKB>PAR=LIBRES
TKB>//
```

base — is the octal byte address that defines the start of the partition. If the library is position-independent (see Section 7.3.1), the base address is zero. If the library is absolute, the base address must be on a 4K word boundary. For example, if the library is always to be positioned beginning at APR 6, you would specify an octal address of 140000.

length — is the octal number of bytes contained in the partition. If the length is omitted or is zero, it is the size of the executable file.

If length is nonzero, and greater than the size of the executable file that the build produced, the Task Builder automatically extends the size of the resident area to make up the difference.

If the executable file size is greater than the length specified, the Task Builder issues the following error message:

```
%TKB--*DIAG*-TASK HAS ILLEGAL MEMORY LIMITS
```

Default

PAR=GEN

Example

See parameter description, above.

Default

None

Example

```
RUN $TKB
TKB>PROG=OBJ1,OBJ2
TKB>/
ENTER OPTIONS:
TKB>RESLIB=DR2:MYLIB/RO:5:200
TKB>//
```

This example causes the library MYLIB on DR2: in the user's account to be mapped read-only beginning at APR 5. The user informs the Task Builder that this library has D-space usage in APR 7.

12.24 RESSUP—Resident Supervisor-Mode Library

The RESSUP option declares that your task intends to access a user owned supervisor-mode library. The term "user owned" means that the library and the symbol definition file associated with it can reside in any directory that you choose. You can specify the directory along with the other portions of the file specification. Do not place comments on the line with RESSUP.

Syntax

RESSUP=filespec/[-]access-code[:apr]

where:

filespec — is the specification identifying the supervisor mode library. The Task Builder expects to find a symbol table file and a task image file with the same file name (filename.STB and filename.TSK) on the device and account specified. You must omit the file type from the specification.

[-] — indicates whether TKB includes mode switching vectors within the user task. If you specify /SV or/SW, TKB includes a 4-word mode switching vector within the address space of the user task for each call to a supervisor-mode library subroutine. If you specify /-SV or /-SW, you must provide your own mode switching vectors. This is useful if your library contains threaded code. Digital recommends, however, using system supplied vectors whenever possible.

access-code — is the code SV (for read-only) or SW (for read/write), indicating the type of access required by your program.

apr — is an integer in the range 0 to 7 that specifies the first supervisor Active Page Register (APR) that you want TKB to reserve for this supervisor-mode library. For position-independent libraries only, you can specify that the default is the lowest available supervisor APR. One supervisor-mode library is required to be at virtual 0 (that is, /SV:0) and must have the CSM (change supervisor mode) dispatcher present together with the completion routines as described in Chapter 9. Most uses would be /SV:0.

12.27 SUPLIB—Resident Supervisor-Mode Library

The SUPLIB option declares that your task intends to access a systemwide supervisor-mode library. The term "systemwide" means that the Task Builder expects to find the supervisor-mode library and the symbol definition file associated with it in the system library account (LB:).

Syntax

SUPLIB=name:[-]access-code[:apr]

where:

name — is the 1- to 6-character name specifying the supervisor mode library. The Task Builder expects to find a symbol table file and a task image file of the same name (name.STB and name.TSK) on the system device and under the account specified by the system logical LB:. If the files do not exist on LB:, you must use the RESSUP option.

[-] — indicates whether TKB includes mode switching vectors within the user task. If you specify :SV or :SW, TKB includes a 4-word mode switching vector within the address space of the user task for each call to a supervisor-mode library subroutine. If you specify :-SV or :-SW, you must provide your own mode switching vectors. Providing your own mode switching vectors is useful if your library contains threaded code. Digital recommends, however, using system supplied vectors whenever possible.

access-code — is the code SV (for read-only) or SW (for read/write), indicating the type of access required by your program.

apr — is an integer in the range 0 to 7 that specifies the first supervisor Active Page Register (APR) that you want TKB to reserve for this supervisor-mode library. You can specify an APR only for position-independent supervisor mode libraries. The default is the lowest available APR. One supervisor-mode library is required to be at virtual 0 (for example, :SV:0) and must have the CSM (change supervisor mode) dispatcher present together with the completion routines as described in Chapter 9. Most uses would be :SV:0.

12.28 TASK—Program Name for SYSTAT

The TASK option lets you specify the name of the program being built. This name is displayed by the SYSTAT program. You can use this option if you want to give a name to a program other than the name of the executable program file.

Syntax

TASK=program-name

where:

program-name	is the one- to six-character name to identify the program in SYSTAT. The characters within the name must be letters (A to Z), numbers (0 to 9), periods (.), or dollar signs ($).

Default

TASK=executable file name

Example

```
 RUN $TKB
TKB> PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB> TASK=USER
TKB> //
```

12.31 VARRAY - Virtual Array Specification and Usage

A virtual array in FORTRAN is a defined area outside of the virtual address space of a task, but within the task's logical address space. The TKB assigns the name $VIRT to the virtual array and automatically provides code to create a dynamic region in memory. Refer to Section 7.7 for more dtails on the use of virtual arrays. The VARRAY=OVR option specifies an overlaid virtual array so that it may be used similarly to the way a FORTRAN COMMON is used. This means that each segment of an overlaid task that uses it defines the array in the same way as it is defined in the root segment. Thereafter, the segment may access the array directly without passing arguments, as is necessary when the array has the concatenated attribute (the default, VARRAY=CON).

To use the VARRAY option with the OVR attribute as VARRAY=OVR, you must first define the array (for example, VIRTUAL DATA(10)), in the root segment of the task. Then, you must define the array in the same way in each segment of an overlaid task using the virtual array. Example 12-1 illustrates how a virtual array may be directly accessed by segments in a task. The example also shows the TKB command line and overlay description file for building the task.

Using the VARRAY option with the CON attribute as VARRAY=CON (the default operation) results in a virtual array subject to the restrictions and uses that are described in the reference manual for the specific kind of FORTRAN that you are using.

Syntax

VARRAY=option

where:

option is either OVR or CON.

Default

VARRAY=CON

Example 12–1: A Task Using a Virtual Array with the OVR Attribute

```
C
C   Program to test the Task Builder option VARRAY
C
      PROGRAM MAIN
      IMPLICIT INTEGER *2 (A-Z)
      VIRTUAL DATA(10)
      CALL INPUT
      CALL CALC
      CALL OUTPUT
      CALL EXIT
      END
```

(continued on next page)

Example 12–1 (Cont.): A Task Using a Virtual Array with the OVR Attribute

```
      SUBROUTINE INPUT
      IMPLICIT INTEGER *2 (A-Z)
      VIRTUAL DATA(10)
      TYPE 10
10    FORMAT (1X,'INPUT I '$)
      ACCEPT 20,DATA(1)
20    FORMAT (I2)
      TYPE 30
30    FORMAT (1X, 'INPUT J '$)
      ACCEPT 20,DATA(2)
      RETURN
      END

      SUBROUTINE CALC
      IMPLICIT INTEGER *2 (A-Z)
      VIRTUAL DATA(10)
      DATA(3) = DATA(1) + DATA(2)        !I + J
      DATA(4) = DATA(1) - DATA(2)        !I - J
      DATA(5) = DATA(1) * DATA(2)        !I * J
      DATA(6) = DATA(1) / DATA(2)        !I / J
      RETURN
      END

      SUBROUTINE OUTPUT
      IMPLICIT INTEGER *2 (A-Z)
      VIRTUAL DATA(10)
      TYPE 10.DATA(3),DATA(4),DATA(5),DATA(6)
10    FORMAT (1X, 'I + J =',I6,/,1X,'I - J =',16,/
      1,1X'I * J=',I6,/,1X.'I / J =',I6)
      RETURN
      END
;
; Command file MAINFT.CMD to build MAINFT.TSK
;
MAINFT/FP,MAINFT/MA/-WI=MAINFT/MP
VARRAY=OVR
//
;
;Overlay description file MAINFT.ODL for MAINFT.TSK
;       .ROOT    $MAIN-*($INPT,$CALC,$OUTP)
$MAIN:  .FCTR    MAIN-LB:[1,1]F77OTS/LB
$INPT:  .FCTR    INPUT-LB:[1,1]F77OTS/LB
$CALC   .FCTR    CALC-LB:[1,1]F77OTS/LB
$OUTP   .FCTR    OUTPUT-LB:[1,1]F77OTS/LB
        .END
```

13.2 The .END Command

Use the .END command as the last line in the ODL file. The .END command tells the Task Builder where the input ends. The format of the .END command is:

.END

13.3 The .FCTR Command

The .FCTR command lets you build large, complex overlay structures and represent them clearly. The format of the .FCTR command is:

label: .FCTR structure

where:

label — at the beginning of the line is used as a part of the structure of a .ROOT or another .FCTR command. The label must be unique with respect to file names and other labels. The structure portion of the .FCTR command can be made up of the same components as the structure of a .ROOT command.

The .FCTR command lets you extend the overlay tree description beyond the one line possible in a .ROOT command. For example:

```
        .ROOT   AFCTR,BFCTR
AFCTR:  .FCTR   A-LIB-(A1-LIB,A2-LIB)
BFCTR:  .FCTR   B-LIB-(B1-LIB,B2FCTR)
B2FCTR: .FCTR   B2-LIB(B21-LIB,B22-LIB,B23-LIB)
LIB:    .FCTR   LB:F4POTS/LB
        .END
```

In the example above, the AFCTR and BFCTR items in the .ROOT command are expanded in following .FCTR commands. Likewise, B2FCTR and LIB are defined in the third and fourth .FCTR commands. The B2FCTR item is defined in a "nested" .FCTR command; that is, the B2FCTR item is defined by a .FCTR command nested within the BFCTR item's defining .FCTR command. The .FCTR command can be nested in this manner to 16 levels.

13.4 The .NAME Command

The .NAME command lets you give a name to a segment and then assign attributes to a segment. As described in Chapter 3, a segment is a piece of your overlay structure that can be loaded in one disk access.

The name you assign must be unique; that is, it must be different than file names, program section names, .FCTR labels, and other segment names used in the overlay description.

The chief purpose of the command is to assign a name to a null co-tree root (Section 4.2), and to make a data segment autoloadable (Section 6.5).

The format of the .NAME command is:

.NAME segment-name[,attr][,attr]

where:

segment-name — is a one- to six-character name consisting of the characters A-Z, 0-9, and $. The name applies to the segment defined immediately following the name when it is used in a .ROOT or .FCTR command. A segment is formed by pieces connected by a dash (-) without intervening parentheses. (Pieces connected by a comma are overlaid and are stored as separate segments.)

attr — is one of the following:

- GBL — Defines the segment-name as a global symbol. As such, it can be referred to in transfer-of-control statements from other pieces of the overlay structure. When such a transfer of control is executed, the segment is loaded, and control is returned to the statement or instruction immediately following the call. Used chiefly in making data segments autoloadable (see Section 6.5).
- NOGBL — Does not define the segment-name as a global symbol. Hence, the name cannot be referred to in transfer-of control statements from other pieces of the overlay structure. If the GBL attribute is not specified, NOGBL is assumed. You would use this attribute when using .NAME to define a null segment as a co-tree root (see Section 4.2).
- NODSK — No disk space is allocated to the named segment in the executable file. If a data overlay segment has no initial values but will be generated by the running program, there is no need to reserve space for it. If you request this option, and the code in your program assigns initial data values to the segment, the Task Builder terminates the build with a fatal error:

  LOAD ADDR OUT OF RANGE IN MODULE file-name
- DSK — Disk space is allocated to the named segment in the executable file. If you do not specify NODSK, DSK is assumed.

If more than one name is applied to a segment, the attributes of the last name given take effect.

13.5 The .PSECT Command

You use the .PSECT command to define the name and attributes of a program section that you want to place in the structure of a .ROOT or .FCTR command. In other words, you can directly specify the placement of a program section named in a .PSECT command.

The general form of the .PSECT command is:

.PSECT p-name[,attr1][,attr2]...[,attr4]

where:

p-name — is the name of the program section (a one- to six-character name consisting of the character A-Z, 0-9, or $).

COMPLEX RELOCATION ERROR - DIVIDE BY ZERO: MODULE filename

A zero divisor was detected in a complex expression. The result of the division was set to zero. (A probable cause is division by a global symbol whose value is undefined. You can set a value for a global symbol with the GBLDEF option to correct this.)

FILE filename ATTEMPTED TO STORE DATA IN VIRTUAL SECTION

You should not get this error running the Task Builder on RSTS/E systems. It diagnoses an error for a capability not used on RSTS/E.

FILE filename HAS ILLEGAL FORMAT

The file named is in an invalid format. This can occur if you try to build a text file, such as a source file. Input files must be compiled or assembled object program files or library files containing compiled or assembled object routines.

ILLEGAL APR RESERVATION

An APR parameter specified in a COMMON, LIBR, SUPLIB, RESCOM, RESSUP, or RESLIB option is outside the range 0-7.

ILLEGAL DEFAULT PRIORITY SPECIFIED

Note that this error relates to the PRI option, which is ignored on RSTS/E systems. The error is returned if you specify an illegal value in the use of the PRI option.

ILLEGAL ERROR-SEVERITY CODE octal-list

System error (no recovery). Please send Digital a Software Performance Report (SPR) with a copy of the message containing the octal-list as printed.

ILLEGAL FILENAME invalid-line

The invalid line printed contains a wildcard (*) in a file specification. You cannot use wildcards in file specifications for the Task Builder.

ILLEGAL GET COMMAND LINE ERROR CODE

System error (no recovery). Please send an SPR to Digital.

ILLEGAL LOGICAL UNIT NUMBER invalid-line

You tried to assign a device (ASG option) to a logical unit number larger than the available number of logical units (UNITS option or the default of 6 if the UNITS option is not specified).

ILLEGAL MULTIPLE PARAMETER SETS invalid-line

You tried to specify more parameters for an option than the option format calls for. See Chapter 12 for the correct format for options.

ILLEGAL NUMBER OF LOGICAL UNITS invalid-line

You cannot specify a logical unit number greater than 14.

ILLEGAL ODT OR TASK VECTOR SIZE

You should not get this error on RSTS/E systems; it diagnoses an error for an option not processed by RSTS/E.

ILLEGAL OVERLAY DESCRIPTION OPERATOR invalid-line

The invalid line printed is an ODL line that contains an operator that the Task Builder does not recognize. This error occurs if the first character in a program section or segment name is a period (.).

ILLEGAL OVERLAY DIRECTIVE invalid-line

The invalid line printed contains an unrecognizable overlay command.

ILLEGAL PARTITION/COMMON BLOCK SPECIFIED

You tried to specify a partition option (PAR) or resident area access option (COMMON, LIBR, RESCOM, RESLIB, RESSUP, OR SUPLIB) defining a resident area as starting not on a 32-word boundary.

ILLEGAL P-SECTION/SEGMENT ATTRIBUTE

You tried to define an attribute for a program section or segment that is not recognizable to the Task Builder. See the description of the .PSECT command or .NAME command in Chapter 13.

ILLEGAL REFERENCE TO LIBRARY P-SECTION psect-name

Your program attempts to refer to a program section name that exists in a run-time system or resident area but has not named the run-time system or area in a COMMON, HISEG, LIBR, RESCOM, RESLIB, RESSUP, SUPLIB or option.

ILLEGAL SWITCH file-specification

The file specification printed contains an illegal switch or switch value.

INCOMPATIBLE OTS MODULE

TKB did not find the Overlay Run-Time System (OTS) module. The OTS modules are part of the system library. This error occurs if you are using an incompatible version of the system library (SYSLIB.OLB).

INCOMPATIBLE REFERENCE TO LIBRARY P-SECTION psect-name

Your program attempts to refer to more storage in a run-time system or resident library than exists in the run-time system or resident library definition.

INCORRECT LIBRARY MODULE SPECIFICATION invalid-line

The invalid line printed names a library routine name with an invalid character. Valid characters are A-Z, 0-9, space, dollar sign ($), or period (.).

INDIRECT COMMAND SYNTAX ERROR invalid-line

The invalid line printed is a command from an indirect file. You must correct the syntax of the command in the indirect file.

INDIRECT FILE OPEN FAILURE invalid-line

The invalid line printed refers to a command input file that could not be located.

INSUFFICIENT PARAMETERS invalid-line

The invalid line printed contains too few parameters. See Part IV for the correct format for command lines, switches, and options.

MODULE filename MULTIPLY DEFINES XFR ADDR IN SEG segment-name

More than one file making up the root has a start address.

MODULE routine-name NOT IN LIBRARY

The Task Builder could not find the routine named on the /LB switch in the library specified.

NO DYNAMIC STORAGE AVAILABLE

The Task Builder needs additional storage for a symbol table and cannot find it. Refer to Appendix E for ways to optimize Task Builder performance.

NO MEMORY AVAILABLE FOR LIBRARY library-name

The Task Builder could not find enough free virtual memory to map the specified run-time system or resident area. Refer to Appendix E for ways to optimize Task Builder performance.

NO ROOT SEGMENT SPECIFIED

You must specify one .ROOT command in the overlay description file.

NO VIRTUAL MEMORY STORAGE AVAILABLE

The maximum allowable size of the Task Builder work file was exceeded. See Section 7.5.6 for suggestions on reducing the size of the work file.

ONLY ONE HISEG MAY BE SPECIFIED

You attempted to specify more than one high segment. The command that generated this error is ignored.

OPEN FAILURE ON FILE filename

An I/O error occurred while the Task Builder was attempting to open the specified file. Try the build again. If you get the same error, see your system manager and report the I/O error.

OPTION SYNTAX ERROR invalid-line

The invalid line printed contains an option that the Task Builder cannot process because it is specified incorrectly. See Chapter 12 for the correct syntax for options.

OVERLAY DIRECTIVE HAS NO OPERANDS

All overlay commands except .END require operands.

OVERLAY DIRECTIVE SYNTAX ERROR invalid-line

The invalid line printed contains a syntax error or refers to a line that contains an error.

PARTITION par-name HAS ILLEGAL MEMORY LIMITS

The partition named is longer than the available address space.

PASS CONTROL OVERFLOW AT SEGMENT segment-name

System error. Please send an SPR to Digital with a copy of the ODL file associated with the error.

PIC LIBRARIES MAY NOT REFERENCE OTHER LIBRARIES

You have tried to build a position-independent resident area that refers to another resident area.

P-SECTION psect-name HAS OVERFLOWED

You have tried to create a program section larger than (32K-32) words.

REQUIRED INPUT FILE MISSING

At least one input file is required for a build.

REQUIRED PARTITION NOT SPECIFIED

You should not get this error on RSTS/E systems. It diagnoses an error for a capability not used on RSTS/E.

RESIDENT LIBRARY HAS INCORRECT ADDRESS ALIGNMENT invalid-line

The invalid line printed specifies a resident area that has one of the following problems:

1. The library refers to another library with invalid address bounds (that is, not on 4K word boundary).
2. The library has invalid address bounds.

RESIDENT LIBRARY MAPPED ARRAY ALLOCATION TOO LARGE invalid-line

The invalid-line displayed contains a reference to a shared region that has allocated too much memory in the task's mapped array area. The total allocation exceeds the system limit; the maximum usable size on RSTS/E is 255K words.

RESIDENT LIBRARY MEMORY ALLOCATION CONFLICT option

One of the following problems has occurred. You tried to specify:

- More than 7 resident areas.
- The same resident area more than once.
- Absolute resident areas whose memory allocations overlay.

ROOT SEGMENT IS MULTIPLY DEFINED invalid-line

The invalid line printed contains the second .ROOT command encountered in an ODL file. Only one .ROOT command is allowed.

SEGMENT seg-name HAS ADDR OVERFLOW: ALLOCATION DELETED

Within a segment, the program attempted to allocate more than (32K-32) words. A map file is produced if it was specified, but no executable file is produced.

TASK HAS ILLEGAL MEMORY LIMITS

You have tried to build a program whose size exceeds the allowable memory size. (This may be the size defined in a PAR option.) If an executable file was produced, delete it.

TASK HAS ILLEGAL PHYSICAL MEMORY LIMITS mapped-array executable-program program extension

The sum of the values displayed—mapped array size, executable program size, and program extension size—exceeds 2.2 million bytes. The quantities are shown as octal numbers in units of 64-byte blocks. Delete any resulting executable program file.

TASK IMAGE FILE filename IS NON-CONTIGUOUS

This error will only occur if your disk is so full that RSTS/E cannot find contiguous space for your program. The file is therefore created noncontiguous; you can otherwise ignore the error message.

TASK REQUIRES TOO MANY WINDOW BLOCKS

The number of address windows required by the program and any resident areas is more than 16. Only 16 are available.

TASK-BUILD ABORTED VIA REQUEST option-line

The option-line printed contains your request to abort the build. You can now retype commands to rerun the Task Builder.

TOO MANY NESTED .ROOT/.FCTR DIRECTIVES invalid-line

The invalid line printed contains a .FCTR command that exceeds the maximum of 16 nested .FCTR commands.

TOO MANY PARAMETERS invalid-line

The invalid line printed contains an option with more parameters than required.

TOO MANY PARENTHESES LEVELS invalid-line

The invalid line printed contains nested parentheses that exceed the maximum of 16 nested parentheses.

TRUNCATION ERROR IN MODULE filename

You tried to load a global value greater than +127 or less than -128 into a byte. Only the low-order eight bits are loaded.

UNABLE TO OPEN WORK FILE

This error can result from several conditions. For example, the device is full, or the work file is assigned to a private pack where you do not have an account, or the work file device is either not mounted or is mounted read-only.

UNBALANCED PARENTHESES invalid-line

The invalid line printed contains unbalanced parentheses. The number of left parentheses must equal the number of right parentheses.

n UNDEFINED SYMBOLS SEGMENT seg-name

The segment named contains n undefined symbols. If you did not request a memory map file, the symbols are also printed at your terminal.

VIRTUAL SECTION HAS ILLEGAL ADDRESS LIMITS option

This error means that the virtual section was declared larger than the limit on RSTS/E.

WORK FILE I/O ERROR

An I/O error occurred during an attempt to refer to data stored by the Task Builder in its work file.

Appendix B

Task Builder Input Data Formats

A compiled or assembled program (.OBJ file), called an object module, consists of variable-length record information. Six record (or block) types are included in the object language. These records guide the Task Builder in the translation of the object language into a task image.

The six record types are:

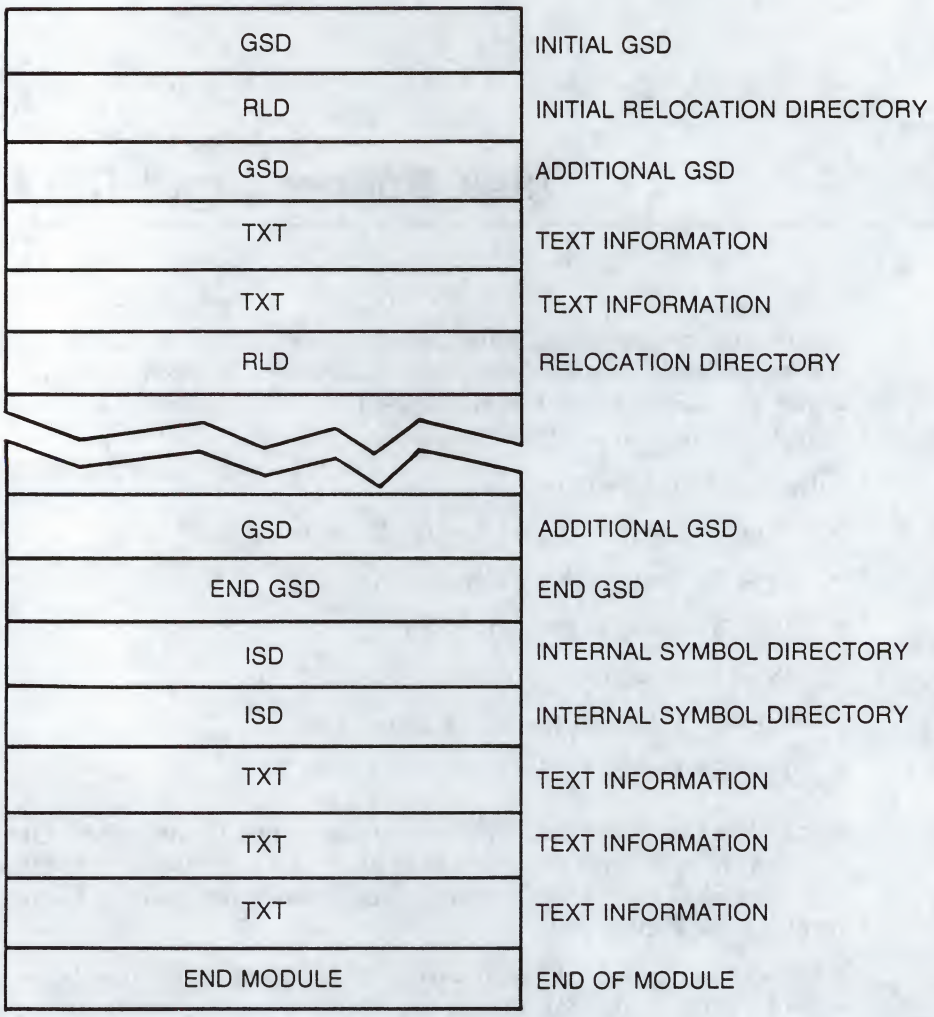
- Type 1 - Declare Global Symbol Directory (GSD)
- Type 2 - End of Global Symbol Directory
- Type 3 - Text Information (TXT)
- Type 4 - Relocation Directory (RLD)
- Type 5 - Internal Symbol Directory (ISD)
- Type 6 - End of Module

Each object module must consist of at least five of the record types. The only record type that is not mandatory is the internal symbol directory. The appearance of the various record types in an object module follows a defined format. See Figure B–1.

An object module must begin with a GSD record and end with an end-of-module record. Additional GSD records can occur anywhere in the file but must appear before an end-of-GSD record. An end-of-GSD record must appear before the end-of-module record, and at least one relocation directory record (RLD) must appear before the first text information record (TXT). Additional RLDs and TXTs can appear anywhere in the file. The internal symbol directory records (ISDs) can appear anywhere in the file between the initial GSD and end-of-module records.

Object module records are of variable length and are identified by a record type code in the first byte of the record. The format of additional information in the record depends on the record type.

Figure B–1: General Object Module Format



B.1 Global Symbol Directory

Global symbol directory (GSD) records contain all the information necessary to assign addresses to global symbols and to allocate the memory required by a task.

GSD records are the only records processed in the first pass. You can save a significant amount of time if you put all GSD records at the beginning of a module, because less of the file must be read on the first pass.

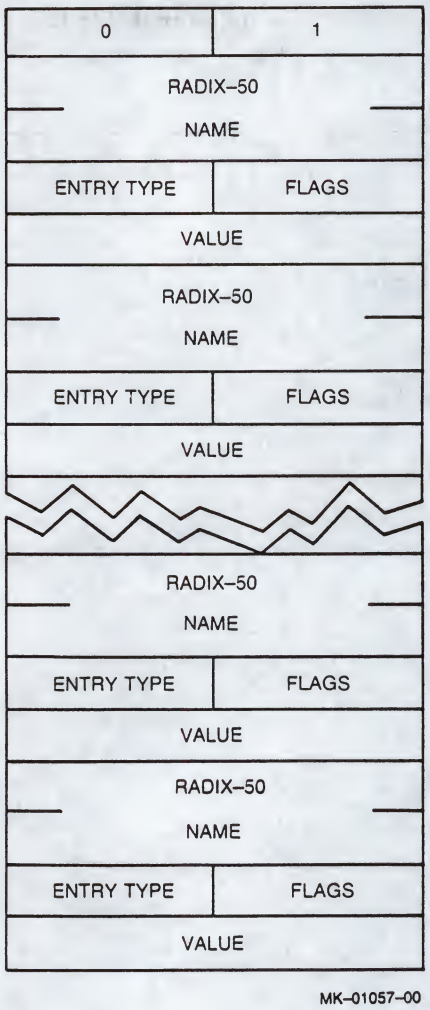
GSD records contain the nine types of entries listed in Table B-1.

Table B–1: GSD Entry Types

Type (Octal)	Entry
0	Module Name
1	Control Section Name
2	Internal Symbol Name
3	Transfer Address
4	Global Symbol Name
5	Program Section Name
6	Program Version Identification
7	Mapped Array Declaration
10	Completion Routine Name

There are four words in the GSD record for each entry type. The first two words contain six Radix-50 characters. The third word contains a flag byte and the entry type identification. The fourth word contains additional information about the entry. See Figure B–2.

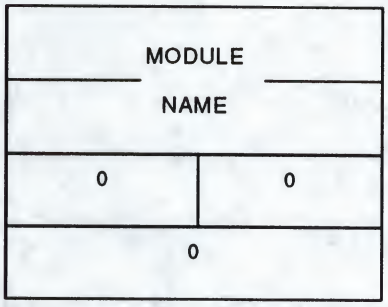
Figure B–2: GSD Record and Entry Format



B.1.1 Module Name

The module name entry, as illustrated in Figure B–3, declares the name of the object module. The name need not be unique with respect to other object modules because modules are identified by file, not module name. Only one module name entry can occur in any given object module.

Figure B–3: Module Name Entry Format



MK–01058–00

B.1.2 Control Section Name

Control sections, which include ASECTs, blank CSECTs, and named CSECTs, are supplanted by PSECTs. For compatibility with other systems, Task Builder processes ASECTs and both forms of CSECTs. Section B.1.6 details the entry generated for a PSECT statement. In terms of the PSECT directive, ASECT and CSECT statements can be defined as follows:

- For a blank CSECT, the PSECT definition is:

  ```
  .PSECT ,LCL,REL,CON,RW,I,LOW
  ```

- For a named CSECT, the PSECT definition is:

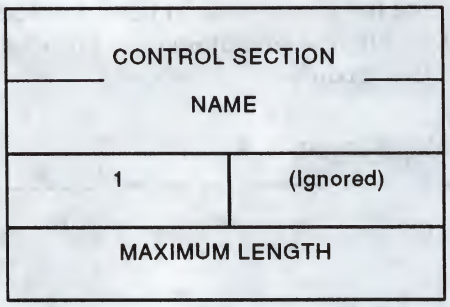
  ```
  .PSECT name,GBL,REL,OVR,RW,I,LOW
  ```

- For an ASECT, the PSECT definition is:

  ```
  .PSECT . ABS.,GBL,ABS,I,OVR,RW,LOW
  ```

ASECTs and CSECTs are processed by the Task Builder as PSECTs with the fixed attributes defined above. The entry generated for a control section is shown in Figure B–4.

Figure B–4: Control Section Name Entry Format

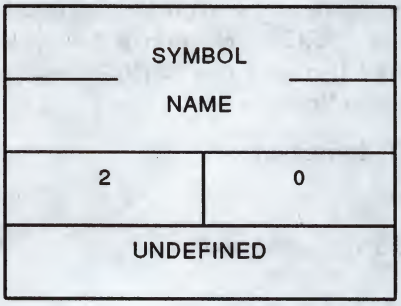


MK–01058–01

B.1.3 Internal Symbol Name

The internal symbol name entry declares the name of an internal symbol (with respect to the module). The Task Builder does not support internal symbol tables, so the detailed format of this entry is not defined (Figure B–5). Any internal symbol entry encountered while the Task Builder reads the GSD is ignored.

Figure B–5: Internal Symbol Name Entry Format

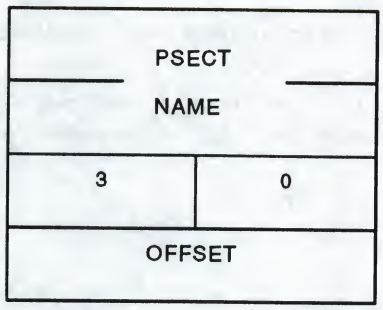


MK–01059–00

B.1.4 Transfer Address

The transfer address entry, as illustrated in Figure B–6, declares the transfer address of a module relative to a PSECT. The first two words of the entry define the name of the PSECT, and the fourth word defines the relative offset from the beginning of that PSECT. If no transfer address is declared in a module, a transfer address entry either must not be included in the GSD or a transfer address 000001 relative to the default absolute PSECT (.ABS.) must be specified.

Figure B–6: Transfer Address Entry Format



MK–01059–01

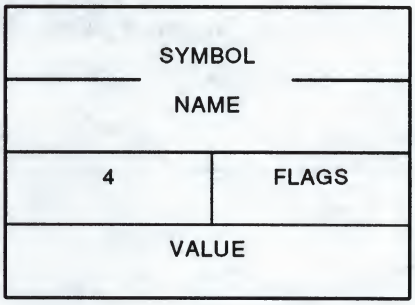
NOTE

If the PSECT is absolute and OFFSET is not 000001, then OFFSET is the actual transfer address.

B.1.5 Global Symbol Name

The global symbol name entry, as illustrated in Figure B–7, declares either a global reference or a definition. All definition entries must appear after the declaration of the PSECT they are defined in and before the declaration of another PSECT. Global references can appear anywhere within the GSD.

Figure B–7: Global Symbol Name Entry Format



MK–01059–02

The first two words of the entry define the name of the global symbol. The flag byte declares the attributes of the symbol, and the fourth word declares the value of the symbol relative to the PSECT it is defined in.

The flag byte of the symbol declaration entry has the following bit assignments:

Bit 0 - Weak Qualifier

0 = Symbol is a strong definition or reference and is resolved in the normal manner.

1 = Symbol is a weak definition or reference. A weak reference (Bit 3 = 0) is ignored. A weak definition (Bit 3 = 1) is ignored unless a previous reference has been made.

Bit 1 - Not used

Bit 2 - Definition Type

0 = Normal Definition of reference.

1 = Library definition. If the symbol is defined in a resident library .STB file, the base address of the library is added to the value, and the symbol is converted to absolute (bit 5 is reset); otherwise, the bit is ignored.

Bit 3 - Reference or Definition

0 = Global symbol reference.

1 = Global symbol definition.

Bit 4 - Not used

Bit 5 - Relocation

0 = Absolute symbol value.

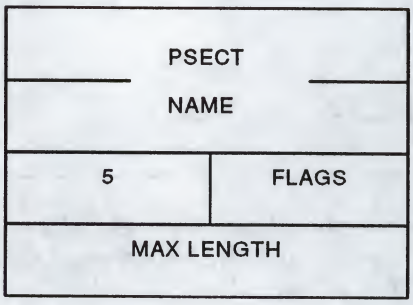
1 = Relative symbol value.

Bit 6-7 - Not used

B.1.6 PSECT Name

The PSECT name entry, as illustrated in Figure B–8, declares the name of a PSECT and its maximum length in the module. It also declares the attributes of the PSECT in the flag byte.

Figure B–8: PSECT Name Entry Format



MK–01060–00

GSD records must be constructed such that, once a PSECT name has been declared, all global symbol definitions pertaining to it must appear before another PSECT name is declared. Global symbols are declared in symbol declaration entries. Thus, the normal format is a series of PSECT names each followed by optional symbol declarations.

The flag byte of the PSECT entry has the following bit assignments:

Bit 0 - SAV PSECT

0 = Normal PSECT.

1 = PSECT is forced into the root of the task.

Bit 1 - Library PSECT

0 = Normal PSECT.

1 = Relocatable PSECT that references a resident library or common block.

Bit 2 - Allocation

0 = PSECT references are to be concatenated with other references to the same PSECT to form the total memory allocated to the PSECT.

1 = PSECT references are to be overlaid. The total memory allocated to the PSECT is the largest request made by individual references to the same PSECT.

Bit 3 - Reserved for the Task Builder

Bit 4 - Access

0 = PSECT has read/write access.

1 = PSECT has read-only access.

Bit 5 - Relocation

0 = PSECT is absolute and requires no relocation.

1 = PSECT is relocatable, and references to the control PSECT must have a relocation bias added before they become absolute.

Bit 6 - Scope

0 = The scope of the PSECT is local. References to the PSECT are collected only within the segment in which the PSECT is defined.

1 = The scope of the PSECT is global. References to the PSECT are collected across segment boundaries. The segment in which a global PSECT is allocated storage is determined either by the first module that defines the PSECT on a path or by direct placement of a PSECT in a segment by the .PSECT directive.

Bit 7 - Type

0 = The PSECT contains instruction (I) references.

1 = The PSECT contains data (D) references.

NOTE

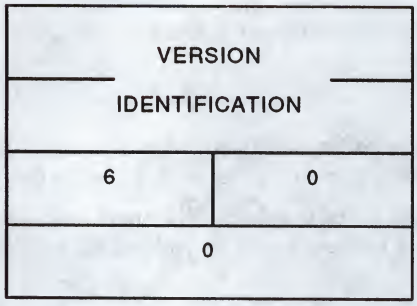
The length of all absolute PSECTs is zero.

B.1.7 Program Version Identification

The program version identification entry, as illustrated in Figure B–9, declares the version of the module. The Task Builder saves the version identification of the first module that defines a nonblank version. This identification is then included on the memory allocation map and is written in the label block of the task image file.

The first two words of the entry contain the version identification. The flag byte and fourth words are not used and contain no meaningful information.

Figure B–9: Program Version Identification Entry Format



MK–01060–01

B.1.8 Mapped Array Declaration (Type 7)

The mapped array declaration entry allocates space within the mapped array area of task memory. The array name is added to the list of task program section names and may be referred to by subsequent RLD records. The length (in units of 64-byte blocks) is added to the task's mapped array location. The total memory allocated to each mapped array is rounded up to the nearest 512-byte boundary. The contents of the flag byte are reserved and assumed to be 0.

One additional window block is allocated whenever a mapped array is declared. Figure B-10 illustrates the mapped array declaration entry format.

Figure B–10: Mapped Array Declaration Entry Format

MAPPED ARRAY	
NAME	
ENTRY TYPE = 7	FLAGS
LENGTH (NUMBER OF 64–BYTE BLOCKS)	

B.1.9 Completion Routine Definition (Type 10)

The completion routine defintion declares the entry point for completion routine of a supervisor-mode library. The data structure is created by the Task Builder and appears only in symbol definition files of supervisor mode libraries.

As shown in figure B-11, the first two words of the entry define the name of the entry point. The third word contains the entry type byte and the flag byte. The flag byte contains no meaningful information. The fourth word contains the symbol value.

Figure B–11: Completion Routine Entry Format

COMPLETION ROUTINE NAME	
ENTRY TYPE = 10	0
VALUE	

B.2 End of Global Symbol Directory

The end-of-global-symbol-directory record, as illustrated in Figure B–12, declares that no other GSD records are contained farther on in the module. Exactly one end-of-GSD record must appear in an object module. Its length is one word.

Figure B–12: End-of-GSD Record Format

0	2

MK–01060–02

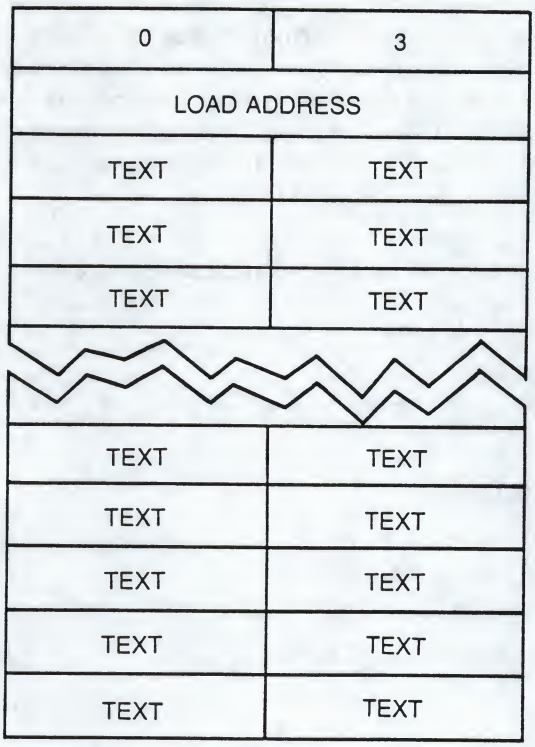
B.3 Text Information

The text information record, as illustrated in Figure B–13, contains a byte string of information that is to be written directly into the task image file. The record consists of a load address followed by the byte string.

Text records can contain words or bytes or a combination of both of information whose final contents have not been determined yet. This information will be bound by a record (see Section B.4). If the test record does not need modification, then no relocation directory record is needed. Thus, multiple text records can appear in sequence before a relocation directory record.

The load address of the text record is specified as an offset from the current PSECT base. At least one relocation directory record must precede the first text record. This directory must declare the current PSECT.

Figure B–13: Text Information Record Format



The Task Builder writes a text record directly into the task image file and computes the value of the load address minus four. This value is stored in anticipation of a subsequent relocation directory that modifies words and bytes that are contained in the test record. When added to a relocation directory displacement byte, this value yields the address of the word and byte to be modified in the task image.

B.4 Relocation Directory

Relocation directory records (see Figure B–14) contain the information necessary to relocate and link the preceding text information record. Every module must have at least one relocation directory record that precedes the first text information record. The first record does not modify a preceding text record but rather defines the current PSECT and location. Relocation directory records contain 15 types of entries. These entries are classified as relocation or location modification entries. Table 7-2 lists the defined types.

Table B–2: Types of Entries for Relocation Directory Records

$Type_8$	Definition
1	Internal Relocation
2	Global Relocation
3	Internal Displaced Relocation
4	Global Displaced Relocation
5	Global Additive Relocation
6	Global Additive Displaced Relocation
7	Location Counter Definition
10	Location Counter Modification
11	Program Limits
12	PSECT Relocation
13	Not used
14	PSECT Displaced Relocation
15	PSECT Additive Relocation
16	PSECT Additive Displaced Relocation
17	Complex Relocation
20	Additive Relocation

Each type of entry is represented by a command byte (specifies type of entry and word/byte modification), followed by a displacement byte, and then by the information required for the particular type of entry. The displacement byte, when added to the value calculated from the load address of the preceding text information record (see Section B.3), yields the virtual address in the image that is to be modified. The command byte of each entry has the following bit assignments:

Bit 0 - 6

Specify the type of entry. Potentially, the 128 command types can be specified, although only 15_{10} are implemented.

Bit 7 - Modification

0 = The command modifies an entire word.

1 = The command modifies only one byte. The Task Builder checks for truncation errors in byte modification commands. If truncation is detected, that is, if the modification value has a magnitude greater than 255, an error occurs.

Figure B–14: Relocation Directory Record Format

0	4
DISP	CMD
INFO	INFO
INFO	INFO
CMD	INFO
INFO	DISP
INFO	INFO
"	"
"	"
"	"
INFO	INFO
DISP	CMD
INFO	INFO
INFO	INFO
INFO	INFO

MK–01062–00

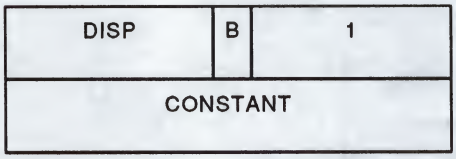
B.4.1 Internal Relocation

The internal relocation entry illustrated in Figure B–15 relocates a direct pointer to an address within a module. The current PSECT base address is added to a specified constant, and the result is written into the task image file at the calculated address. (That is, a displacement byte is added to the value calculated from the load address of the preceding text block.)

For example:

```
A:      MOV         #A,RO
        or
        .WORD       A
```

Figure B–15: Internal Relocation Entry Format



MK-01063-00

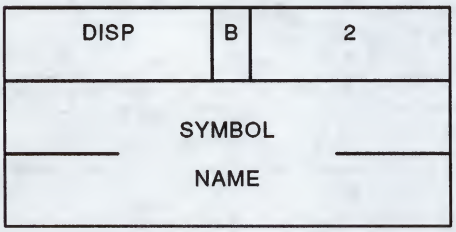
B.4.2 Global Relocation

The global relocation entry in Figure B–16 relocates a direct pointer to a global symbol. The definition of the global symbol is obtained and the result is written into the task image file at the calculated address.

For example:

```
        MOV         #GLOBAL,RO
        or
        .WORD       GLOBAL
```

Figure B–16: Global Relocation Entry Format



MK-01063-01

B.4.3 Internal Displaced Relocation

The internal displaced relocation entry in Figure B–17 relocates a relative reference to an absolute address from within a relocatable control section. The address plus 2 that the relocated value is to be written into is subtracted from the specified constant. The result is then written into the task image file at the calculated address.

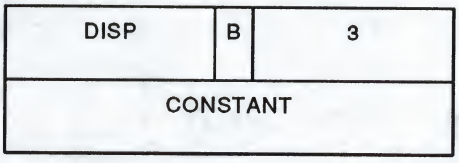
For example:

```
CLR     177550
```

or

```
MOV     177550,RO
```

Figure B–17: Internal Displaced Relocation Entry Format



MK–01063–02

B.4.4 Global Displaced Relocation

The global displaced relocation entry in Figure B–18 relocates a relative reference to a global symbol. The definition of the global symbol is obtained, and the address plus 2 that the relocated value is to be written into is subtracted from the definition value. The result is then written into the task image file at the calculated address.

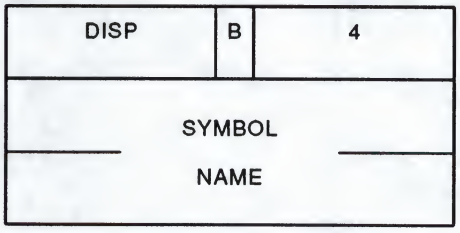
For example:

```
CLR     GLOBAL
```

or

```
MOV     GLOBAL,RO
```

Figure B–18: Global Displaced Relocation Entry Format



MK–01063–03

B.4.5 Global Additive Relocation

The global additive relocation entry in Figure B–19 relocates a direct pointer to a global symbol with an additive constant. The definition of the global symbol is obtained, the specified constant is added, and the resultant value is then written into the task image file at the calculated address.

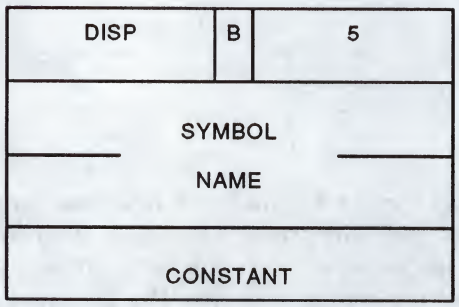
For example:

```
MOV          #GLOBAL+2,RO
```

or

```
.WORD        GLOBAL-4
```

Figure B–19: Global Additive Relocation Entry Format



MK-01064-00

B.4.6 Global Additive Displaced Relocation

The global additive displaced relocation entry in Figure B–20 relocates a relative reference to a global symbol with an additive constant. The definition of the global symbol is obtained, and the specified constant is added to the definition value. The address plus 2 that the relocated value is to be written into is subtracted from the resultant additive value. The result is then written into the task image file at the calculated address.

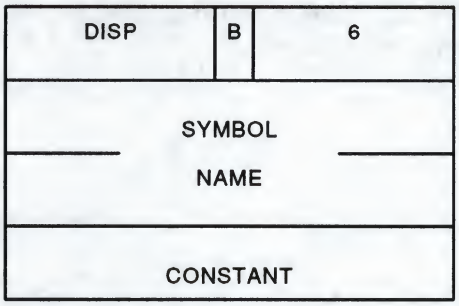
For example:

```
CLR          GLOBAL+2
```

or

```
MOV          GLOBAL-5,RO
```

Figure B–20: Global Additive Displaced Relocation Entry Format

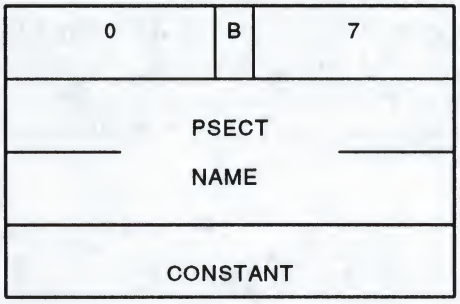


MK–01064–01

B.4.7 Location Counter Definition

The location counter definition in Figure B–21 declares a current PSECT and location counter value. The control base is stored as the current control section, and the current control section base is added to the specified constant and stored as the current location counter value.

Figure B–21: Location Counter Definition



MK–01064–02

B.4.8 Location Counter Modification

The location counter modification entry in Figure B–22 modifies the current location counter. The current PSECT base is added to the specified constant and the result is stored as the current location counter.

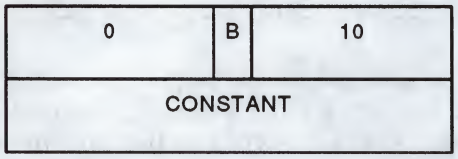
For example:

```
.=.+N
```

or

```
.BLKB        N
```

Figure B–22: Location Counter Modification



MK–01065–00

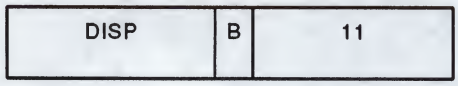
B.4.9 Program Limits

The program limits entry in Figure B–23 is generated by the .LIMIT assembler directive. The first address above the header (normally the beginning of the stack) and the highest address allocated to the task are obtained and written into the task image file at the calculated address and at the calculated address plus 2 respectively.

For example:

```
.LIMIT
```

Figure B–23: Program Limits Entry Format



MK–01065–01

B.4.10 PSECT Relocation

The PSECT relocation entry in Figure B–24 relocates a direct pointer to the beginning address of another PSECT (other than the PSECT in which the reference is made) within a module. The current base address of the specified PSECT is obtained and written into the task image file at the calculated address.

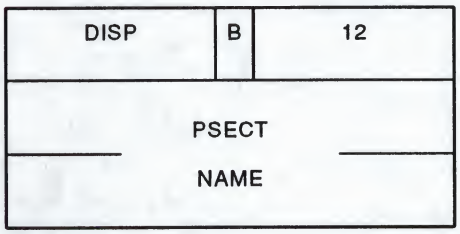
For example:

```
        .PSECT   A
B:
        .
        .
        .
        .PSECT   C
        MOV      #B,RO
```

or

```
        .WORD    B
```

Figure B–24: PSECT Relocation Entry Format



MK-01065-02

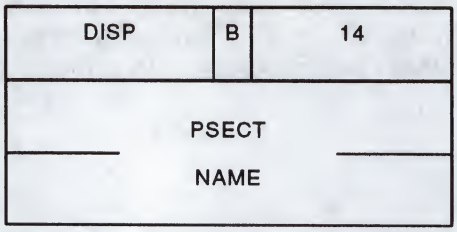
B.4.11 PSECT Displaced Relocation

The PSECT displaced relocation entry in Figure B–25 relocates a relative reference to the beginning address of another PSECT within a module. The current base address of the specified PSECT is obtained and the address plus 2 that the relocated value is to be written into is subtracted from the base value. The result is then written into the task image file at the calculated address.

For example:

```
        .PSECT   A
B:
        .
        .
        .
        .PSECT   C
        MOV      B,RO
```

Figure B–25: PSECT Displaced Relocation Entry Format



MK–01066–00

B.4.12 PSECT Additive Relocation

The PSECT additive relocation entry in Figure B–26 relocates a direct pointer to an address in another PSECT within a module. The current base address of the specified PSECT is obtained and added to the specified constant. The result is written into the task image file at the calculated address.

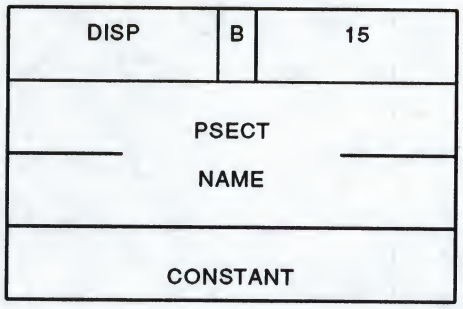
For example:

```
        .PSECT   A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT   D
        MOV      #B+10,RO
        MOV      #C,RO
```

or

```
        .WORD    B+10
        .WORD    C
```

Figure B–26: PSECT Additive Relocation Entry Format



MK–01066–01

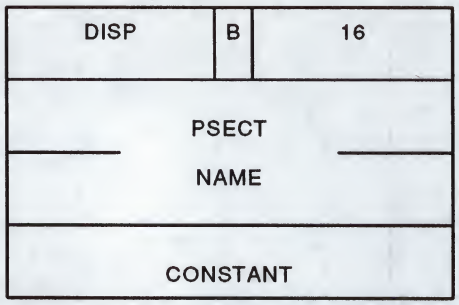
B.4.13 PSECT Additive Displaced Relocation

The PSECT additive displaced relocation entry in Figure B–27 relocates a relative reference to an address in another PSECT within a module. The current base address of the specified PSECT is obtained and added to the specified constant. The address plus 2 that the relocated value is to be written into is subtracted from the resultant additive value. The result is then written into the task image file at the calculated address.

For example:

```
        .PSECT   A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT   D

        MOV      B+10,RO
        MOV      C,RO
```

Figure B–27: PSECT Additive Displaced Relocation Entry Format



MK–01066–02

B.4.14 Complex Relocation

The complex relocation entry in Figure B–28 resolves a complex relocation expression. In such an expression, any of the MACRO-11 binary or unary operations are permitted. Any type of argument is permitted, regardless of whether the argument is unresolved global, relocatable to any PSECT base, absolute, or a complex relocatable subexpression.

The RLD command word is followed by a string of numerically-specified operation codes and arguments. Each operation code occupies one byte. The entire RLD command must fit in a single record. Table B-3 lists the defined operation codes.

Table B–3: Defined Operation Codes for the RLD Command Word

0	No operation.
1	Addition (+).
2	Subtraction (-).
3	Multiplication (*).
4	Division (/).
5	Logical AND (&).
6	Logical inclusive OR (!).
10	Negation (-).
11	Complement ^C.
12	Store result (command termination).
13	Store result with displaced relocation (command termination).
16	Fetch global symbol. It is followed by four bytes containing the symbol name in Radix-50 representation.
17	Fetch relocatable value. It is followed by one byte containing the sector number and two bytes containing the offset within the sector.
20	Fetch constant. It is followed by two bytes containing the constant.

(continued on next page)

Table B–3 (Cont.): Defined Operation Codes for the RLD Command Word

21	Fetch resident library base address. If the file is a resident library .STB file, the library base address is obtained; otherwise, the base address of the Task Image is fetched.

The STORE commands indicate that the value is to be written into the task image file at the calculated address.

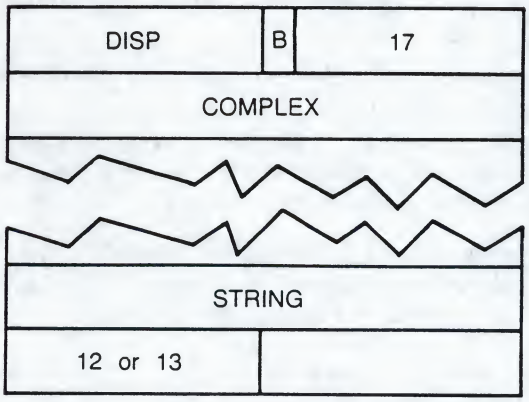
All operands are evaluated as 16-bit signed quantities using two's complement arithmetic. The results are equivalent to expressions that are evaluated internally by the assembler. Note the following rules:

1. An attempt to divide by zero yields a zero result. The Task Builder issues a nonfatal diagnostic message.
2. All results are truncated from the left in order to fit into 16 bits. No diagnostic message is issued if the number was too large. If the result modifies a byte, the Task Builder checks for truncation errors as described in Section B.4.
3. All operations are performed on relocated (additive) or absolute 16-bit quantities. PC displacement is applied to the result only.

For example:

```
            .PSECT ALPHA
A:
            .
            .
            .
            .PSECT BETA
B:
            .
            .
            .
              MOV     #A+B -<G1/G2&^C<177120!G3>>,R1
```

Figure B–28: Complex Relocation Entry Format



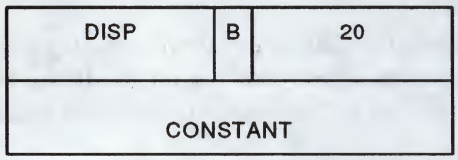
MK–01067–00

B.4.15 Additive Relocation

The shared run-time system (SRTS) additive relocation entry in Figure B–29 relocates a direct pointer to an address within an SRTS.

If the current file is a symbol table file (STB), the base address of the SRTS is obtained and added to the specified constant. The result is written into the task image file at the calculated address. If the file is not associated with an SRTS, the task base address is used.

Figure B–29: Additive Relocation Entry Format



MK–01067–01

B.5 Internal Symbol Directory Record

The Internal Symbol Directory (ISD) record declares definitions of all symbols that are defined in the module. In addition to looking for global symbol definitions in the input object modules, TKB must look for ISD records. Some of these require no relocation and TKB can copy them directly into the .STB file. Others will require modification; after being modified, ISD records can be written to the .STB file. In addition, TKB may need to generate some ISD records of its own in the .STB file.

Except for autoloadable library entry points, TKB puts ISD records into the .STB file only if the /DA switch is used in the TKB command line. When TKB outputs the .STB file, it writes one of three major types of ISD records:

- Type 1 records, where TKB generates ISDs in language-independent form.
- Type 3 records, written for any type 2 records in an input object module. TKB does this after adding data and then changing the ISD record type to language-dependent and independent sections. Language processors generate these records and TKB modifies them. They contain information that can be used to find the absolute task image address of source program entities (for example, variables, program statements, and so on).
- Type 4 records, written to the .STB file without modification. Type 4 records are literal records that contain language-dependent information. Apart from the first few bytes, TKB ignores the rest of the record.

The following sections describe the record formats.

B.5.1 Overall Record Format

ISD records have the same basic structure as all object language records. Because of the variety of different types, the skeleton structure must include additional fields that are common to all ISD record types. The general format of all ISD records is shown in Figure B–30.

Figure B–30: General Format of All ISD Records

MUST BE 0	RECORD TYPE = 5
RESERVED (0)	ISD RECORD TYPE
RECORD TYPE DEPENDENT	

MK–01068–00

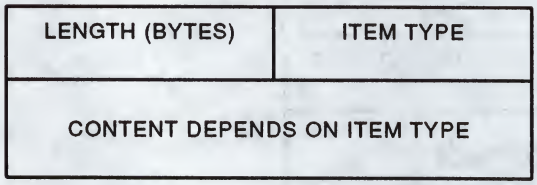
ISD record types fall into these general categories:

0	Illegal.
1	TKB-generated.
2	Compiler-generated relocatable.
3	Relocated (type 2 after TKB processing).
4-127	Not defined, reserved for future use.
128-255	Literal records. (The type code identifies the generating language processor, and thus, the internal structure.)

B.5.2 TKB-Generated Records (Type 1)

The content of this record type is a string of individual items, each with its own format. The items are either start-of-segment items, task identification items, or autoloadable entry point items. The TKB-generated record is similar to the structure of an RLD or GSD record. The general format is shown in Figure B–31.

Figure B–31: General Format of a TKB-Generated Record

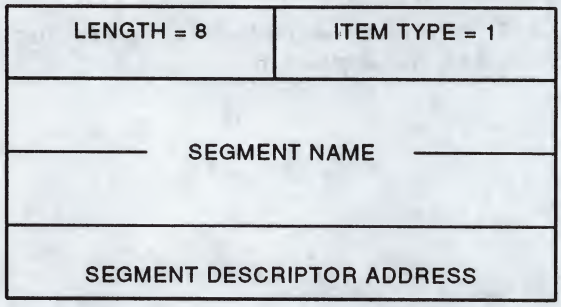


MK–01069–00

B.5.2.1 Start-of-Segment Item (Type 1)

The format of the start-of-segment item type is shown in Figure B–32.

Figure B–32: Format of TKB-Generated Start-of-Segment Item (1)



MK–01070–00

B.5.2.2 Task Identification Item (Type 2)

The task identification item type ensures that an .STB file and the task image being debugged were generated at the same time. Otherwise, symbols that are found may not correspond to the actual task.

The task identification item type exists to make the correlation between the .STB file and its related task possible. The contents of this item type correspond exactly to the first ten words of an area in a task image file, which is in the TKB-created PSECT called $$DBTS.

The format of the task identification item type is shown in Figure B–33.

Figure B–33: Format of TKB-Generated Task Identification Item (2)

LENGTH = 22	ITEM TYPE = 2
EIGHT–WORD TIME STAMP (1)	
TWO–WORD NUMBER (2)	

(1) Its form is that which is returned by RSX directive GTM$.

(2) TKB generates this number as an additional check on correspondence. Currently always zero.

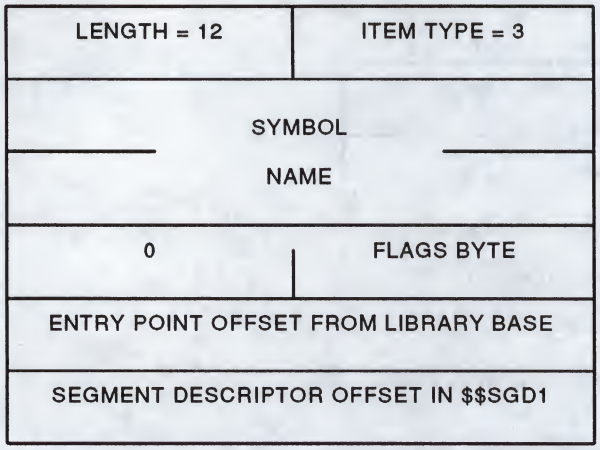
MK–01071–00

B.5.2.3 Autoloadable Library Entry Point Item (Type 3)

TKB outputs the autoloadable library entry point item into an .STB file when building overlaid resident libraries. The ISD record contains the information needed by TKB to dynamically generate autoload vectors for entry points in the library. Autoload vectors appear for only those entry points that are referenced by the task. Unlike the other items, the autoloadable library entry point item is not for use by debuggers.

The format of the autoloadable entry point item is shown in Figure B–34.

Figure B–34: Format of an Autoloadable Library Entry Point Item (3)



MK–01072–00

B.5.3 Relocatable/Relocated Records (Type 2)

These records are the central part of TKB's involvement in debugger communication. Every item type in these records must be standardized, and only standard items can appear. The general format is the same as that shown in Figure B–30.

A language processor outputs these record types as type 2. When TKB processes them, it changes the type to type 3. It also fills in or modifies some fields. In the following descriptions, fields that are filled in by TKB are marked with an asterisk (*). They should be left as zero in language processor output.

B.5.3.1 Module Name Item (Type 1)

A module name item should be the first ISD entry of each object module. A debugger can assume that all following ISD information up to the next module name item relates to this module.

The language code is included so that a debugger for a specific language can determine whether to ignore a module if it is written for another language. The language code has the same range of values as that of a language-dependent ISD record (128-255) and has the same meanings.

The format of the module name item type is shown in Figure B–35.

Figure B–35: Format of a Module Name Item (Type 1)

LENGTH	ITEM TYPE = 1
MUST BE 0	LANGUAGE CODE
MODULE NAME (1)	

(1) A counted ASCII string of the required length. (A counted ASCII string is a byte in which the first byte indicates the number of bytes to follow.)

MK–01073–00

B.5.3.2 Global Symbol Item (Type 2)

One type 2 item should appear for each global symbol definition that the language processor wants the debugger to understand. It need not, for example, include definitions generated for the language processor run-time system.

The format of the global symbol item type is shown in Figure B–36.

Figure B–36: Format of a Global Symbol Item (Type 2)

LENGTH	ITEM TYPE = 2
SYMBOL NAME (RADIX–50)	
VALUE*	
DESCRIPTOR ADDRESS FOR CONTAINING OVERLAY SEGMENT*	
MUST BE ZERO	FLAGS
FULL SYMBOL NAME (1)	

(1) Counted ASCII string of the required length. (A counted ASCII string is a byte string in which the first byte indicates the number of bytes to follow.)

MK–01074–00

B.5.3.3 PSECT Item (Type 3)

A concatenated PSECT has two base addresses: one for the whole PSECT, and the other for the part of it that belongs to this module. It is the base address for the part that belongs to this module that may be used by a debugger to convert local symbol values to absolute addresses.

The segment descriptor address is necessary because a PSECT may move to segments other than the one in which it is placed. This address is relevant to languages that provide semi-automatic overlay generation, like COBOL-11. This word may be zero if the PSECT has not moved to another segment.

The flag word is a copy of the flag word built by TKB. It allows for identification of VSECTs.

Some languages may need the full PSECT name.

The format of a PSECT item type is shown in Figure B–37.

Figure B–37: Format of a PSECT Item (Type 3)

LENGTH	ITEM TYPE = 3
PSECT NAME	
BASE ADDRESS OF PSECT IN THIS SEGMENT*	
BASE ADDRESS OF PSECT FOR THIS MODULE*	
LENGTH OF PSECT FOR THIS MODULE*	
DESCRIPTOR ADDRESS FOR CONTAINING SEGMENT*	
FLAG WORD*	
FULL PSECT NAME (1)	

(1) Counted ASCII string of the required length. (A counted ASCII string is a byte string in which the first byte indicates the number of bytes to follow.)

MK–01075–00

B.5.3.4 Line-number or PC Correlation Item (Type 4)

This item provides the information needed to translate a source line-number into a task image address, or a task image address into a source line-number.

The format of a line-number of PC correlation item type is shown in Figure B–38.

Figure B–38: Format of a Line-Number or PC Correlation Item (Type 4)

LENGTH	ITEM TYPE = 4
PSECT NAME	
START PC (1)	
DESCRIPTOR ADDRESS OF CONTAINING OVERLAY SEGMENT*	
START PAGE NUMBER	
START LINE NUMBER	
STRING OF ONE–BYTE ITEMS	

(1) Offset into PSECT in type 2 records; absolute address in type 3 records.

MK–01076–00

B.5.3.5 Internal Symbol Name Item (Type 5)

It is necessary to allow for the fact that a name may have more than one associated address. For example, a COBOL variable may have three associated addresses: the address of the area that actually contains the data, the address of a CIS descriptor, and the address of a picture string.

The internal symbol name item, which meets these requirements, is shown in Figure B–39.

Figure B–39: Format of an Internal Symbol Name Item (Type 5)

	LENGTH	ITEM TYPE = 5
	OFFSET TO NAME	OFFSET TO DATA
	MUST BE ZERO	NUMBER OF ADDRESSES
ADDRESS 1:	PSECT NAME	
	TASK IMAGE ADDRESS/OFFSET (1)	
	SEGMENT DESCRIPTOR ADDRESS*	
ADDRESS 2:	PSECT NAME	
	TASK IMAGE ADDRESS/OFFSET (1)	
	SEGMENT DESCRIPTOR ADDRESS*	
ADDRESS n:	. . . . .	. . . . .
	LANGUAGE-DEPENDENT DATA	
	SYMBOL NAME (2)	

(1) Modified by TKB.

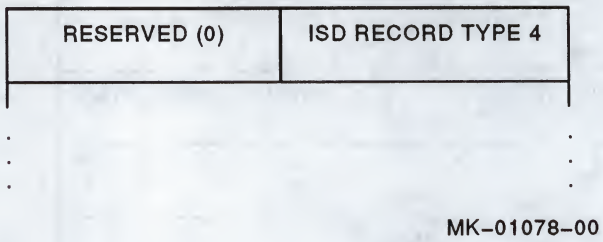
(2) A counted ASCII string of the required length. (A counted ASCII string is a byte string in which the first byte indicates the number of bytes to follow.

MK-01077-00

B.5.4 Literal Records (Type 4)

Literal records may take any form except for the two-byte header shown in Figure B–40.

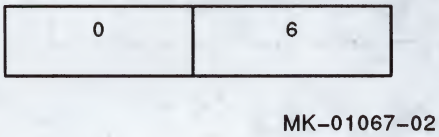
Figure B–40: Format of a Literal Record Type



B.6 End of Module

The end-of-module record in Figure B–41 declares the end of an object module. Exactly one end-of-module record must appear in each object module. It is one word in length.

Figure B–41: End-of-Module Record Format



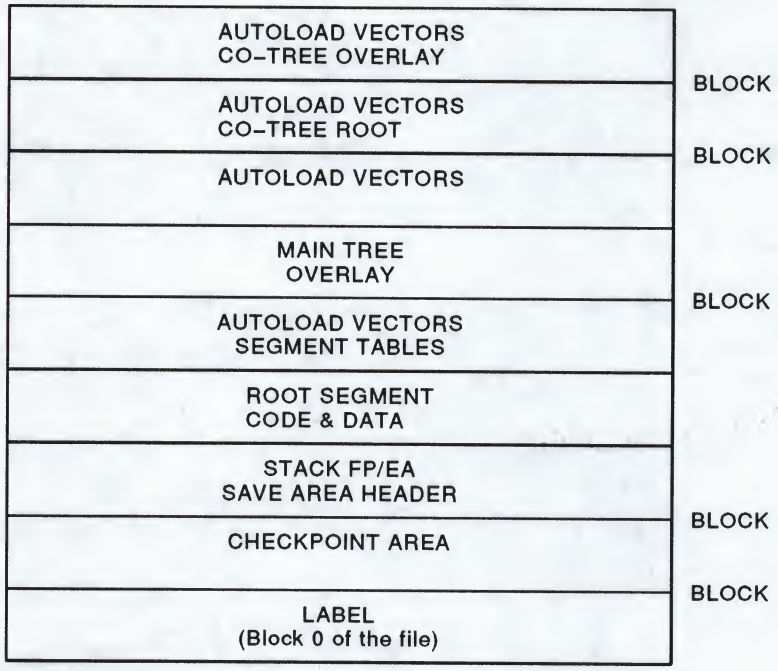
Appendix C

Executable File Structure

This appendix describes the elements of an executable file.

The executable file as it is recorded on the disk appears in Figure C–1.

Figure C–1: Task Image on Disk



MK–01079–00

C.1 Label Block Group

The label block group shown in Figure C–2 precedes the task on the disk and contains data that need not be resident during task execution. This group is composed of two elements:

- Task and resident library data (Label Block 0)
- Table of LUN assignments (Label Block 1) which contains the name and logical unit number of each device assigned

Figure C–2: Label Block Group

Label	Non–I&D Tasks/ID Tasks	Offset *		
L$BTSK		0	Task	
		2	Name	
L$BPAR		4	Task	
		6	Partition	
L$BSA		10	Base address of task	
L$BHGV		12	Highest window 0 virtual address	
L$BMXV		14	Highest virtual address in task	
L$BLDZ		16	Load size in 64–byte blocks	
L$BMXZ		20	Maximum size in 64–byte blocks	
L$BOFF		22	Task offset into partition	
L$BWND/L$BSYS		24	System ID | Number of window blocks*	
L$BSEG		26	Size of overlay segment descriptors	
L$BFLG		30	Task flag word	
L$BDAT		32	Task creation date –Year	
		34	– Month	
		36	– Day	
L$BLIB		40	Library/common	
		42	Name	R$LNAM
		44	Base address of library	R$LSA
		46	Highest address in first library window	R$LHGV
		50	Highest address in library	R$LMXV
		52	Library load size (64–byte blocks)	R$LLDZ
		54	Library maximum size (64–byte blocks)	R$MXZ
		56	Library offset into region	R$LOFF
		60	Number of library window blocks	R$LWND
		62	Size of library segment descriptors	R$LSEG
		64	Library flag word	R$LFLG
		66	Library creation date – Year	R$LDAT
		70	– Month	
		72	– Day	
		⋮	⋮	
		344/704	0	
L$BPRI		346/706	Task priority	
L$BXFR		350/710	Task transfer address	
L$BEXT		352/712	Task extension (64–byte blocks)	
L$BSGL		354/714	Block number of segment load list	
L$BHRB		356/716	Block number of header	
L$BBLK		360/720	Number of blocks in label	
L$BLUN		362/722	Number of logical units	
L$BROB		364/724	Relative block of R–O image	
L$BROL		366/726	R/O load size	
L$BRDL		370/730	R/O data size in 32–word blocks	
L$BHDB		372/732	Relative block number of data header	
L$BDHV		374/734	High virtual address of data window 1	
L$BDMV		376/736	High virtual address of data	
L$BDLZ		400/740	Load size of data	
L$BDMZ		402/742	Mazimum size of data	
L$BAPR		404/744	APR mask word	
L$DAPR		404/772	Second task flag word	
L$BFLZ		404/774	Label block revision number	
L$BLRL		404/776	AME (must be 0)	

Library Request (maximum of 7 or 15 14–word entries)

* Less library window blocks.

* If the image is a SIL (output of MAKSIL program), add 1000 (octal) to these values. Also, location 776 (no 1000 added)=SIL in Radix–50.

Task and resident library data are described in Table C-1.

Table C–1: Task and Resident Library Data

L$BTSK	Task name consisting of two words in Radix-50 format. This parameter is set by the TASK keyword.
L$BPAR	Partition name consisting of two words in Radix-50 format. This parameter is set by the PAR keyword.
L$BSA	Starting address of task. Marks the lowest task virtual address. This parameter is set by the PAR keyword.
L$BHGV	Highest virtual address mapped by address window 0.
L$BMXV	Highest task virtual address. This value is set to L$BHGV.
L$BLDZ	Task load size in units of 64-byte blocks. This value represents the size of the root segment.
L$BMXZ	Task minimum size in units of 64-byte blocks. This value represents the size of the root segment plus any additional physical memory needed to contain task overlays.
L$BOFF	Task offset into partition in units of 64-byte blocks.
L$BWND	Number of task windows (less windows of declared libraries (SRTS))- Low byte.
L$BSYS	System I.D.- High byte 1 = RSX-11M. 4 = RSX-11M-PLUS.
L$BSEG	Size of overlay segment descriptors (in bytes).
L$BFLG	Task flags word. The following flags are defined:

Bit	Flag	Meaning When Bit = 1
15	TS$PIC	Task contains position-independent code (PIC).
14	TS$NHD	Task has no header.
12	TS$PMD	Task generates Postmortem Dump.
7	TS$CMP	Task is built in compatibility mode.
6	TS$CHK	Task is not check-pointable (not supported on RSTS/E).
5	TS$RES	Task has memory-resident overlays.
3	TS$SUP	Image linked as supervisor-mode library.
0	TS$NEW	New header format L$BFL2 and after are valid.

L$BDAT	Three words containing the task creation date as two-digit integer values: Year (since 1900) Month of year Day of month
L$BLIB	Resident library entries.
L$BPRI	Task priority set by the PRI keyword. Ignored by RSTS/E.
L$BXFR	Task transfer address. (Not used by RSTS/E.)
L$BEXT	Task extension size in units of 64-byte blocks. This parameter is set by the EXTTSK keyword.
L$BSGL	Relative block number of segment load list. Set to zero if no list is allocated.

(continued on next page)

Table C–1 (Cont.): Task and Resident Library Data

L$BHRB	Relative block number of header.
L$BBLK	Number of blocks in label block group.
L$BLUN	Number of logical units.
L$BROB	Relative block number of R/O image.
L$BROL	R/O load size in 32-word blocks.
L$BRDL	Size of R/O data in 32-word blocks.
L$BHDB	Relative block number of data header.
L$BDHV	High virtual address of data window 1 of D-space task.
L$BDMV	High virtual address of data.
L$BDLZ	Load size of data.
L$BDMZ	Maximum size of data.
L$BAPR	The APR mask word.
L$BFL2	Second task flag word. The following flags are defined:
	Bit — Flag — Meaning When Bit = 1
	1 — TZ$FMP — Task uses fast map facility.
L$BLRL	Label block revision level.
L$AME	Always 0 for AME compatibility.

The contents of the SRTS/common name block are listed in Table C-2. This block is constructed by referencing the disk image of the SRTS/common block. The format is identical to words 3 through 16 of the label block.

Table C–2: Contents of SRTS/Common Name Block

R$LNAM	Library/command name consisting of two words in Radix-50 format.
R$LSA	Base virtual address of library or common.
R$LHGV	Highest address mapped by first library window.
R$LMXV	Highest virtual address in library or common.
R$LLDZ	Library/common block load size in 64-byte blocks.
R$LMXZ	Library maximum size in units of 64-byte blocks.
R$LOFF	Size of mapped array allocated by the resident library.
R$LWND	Number of window blocks required by library.
R$LSEG	Size of library overlay segment descriptors in bytes.
R$LFLG	Library flags word. The following flags are defined: Bit — Meaning 15 — LD$ACC - Access intent (1=read/write, 0=read-only) 14 — LD$RSV - APR was reserved 13 — LD$CLS - Library is part of a cluster 7 — Default member of cluster (or HISEG). 5 — LD$RES-library has memory resident overlays. 3 — LD$SUP - Supervisor-mode library (1=yes) 2 — LD$REL - Position-independent code (PIC) flag (1=PIC) 1 — LD$TYP-shared region type (1=common, 0=library).
R$LDAT	Three words containing the library/common block creation date in the following format: WORD 0: Year since 1900 WORD 1: Month of year WORD 2: Day of month

C.2 Header

The task header starts on a block boundary and is immediately followed by the task image. The task is read into memory starting at the base of the root segment. Because the root segment is a set of contiguous disk blocks, it is loaded with a single disk access.

The header is divided into two parts: a fixed part, as shown in Figure C–3, and a variable part, as shown in Figure C–4.

Figure C–3: Task Header Fixed Part

H.CSP	0	Current Stack Pointer (R6)
H.HDLN	2	Header length
H.EFLM	4	Event flag mask
	6	Event flag address
H.CUIC	10	Current UIC
H.DUIC	12	Default UIC
H.IPS	14	Initial PS
H.IPC	16	Initial PC (R7)
H.ISP	20	Initial Stack Pointer (R6)
H.ODVA	22	ODT SST vector address
H.ODVL	24	ODT SST vector length
H.TKVA	26	Task SST vector address
H.TKVL	30	Task SST vector length
H.PFVA	32	Power fail AST control block
H.FPVA	34	Floating Point AST control block
H.RCVA	36	Receive AST control block
H.EFSV	40	Address of event flag context
H.FPSA	42	Address of floating point context
H.WND	44	Pointer to number of window blocks
H.DSW	46	Directive Status Word
H.FCS	50	Address of FCS impure storage
H.FORT	52	Address of language impure storage
H.OVLY	54	Address of overlay impure storage
H.VEXT	56	Address of impure vectors
H.SPRI H.NML	60 61	Swapping priority Mailbox LUN
H.RRVA	62	Receive by reference AST control block
	64	Reserved
	66	Reserved
	70	Reserved
H.GARD	72	Header guard word pointer
H.NLUN	74	Number of LUNs

MK–01081–00

Figure C–4: Task Header Variable Part

H.LUN	LUN Table (2 words/LUN)	
	. . .	
	Number of Window Blocks	Offsets
	Partition Control Block Address	W.BPCB
	Low Virtual Address Limit	W.BLVR
	High Virtual Address Limit	W.BHVR
	Address of Attachment Descriptor	W.BATT
	Window Size (in 32–word blocks)	W.BSIZ
	Offset into Partition (in 32–word blocks)	W.BOFF
	First PDR Address	W.BFPD
	Number of PDRs to Map	W.BNPD
	Contents of Last PDR	W.BLPD
	. . .	

Current PS	
Current PC	INITIAL VALUES
Current R5	relative block number of header
Current R4	indent word #2
Current R3	indent word #1
Current R2	task name word #2
Current R1	task name word #1
Current R0	program transfer address
Header Guard Word	

MK–01082–00

The variable part of the header contains window blocks that describe the following:

- The task's virtual-to-physical mapping
- Logical unit data
- Task context

The task header is used by RSTS/E mainly for setting the initial conditions of the task. Only locations 46 through 56 have identical meanings as in RSX.

NOTE

To save the identification, the initial value set by the Task Builder should be moved to local storage. When the program is fixed in memory and being restarted without reloading, the reserved program words must be tested for their initial values to determine whether the contents of R3 and R4 should be saved.

The contents of R0, R1, and R2 are set only when a debugging aid is present in the task image.

C.2.1 Low Core Context

The low core context for a task consists of the Directive Status Word and the Impure Area vectors. The Task Builder recognizes the following global names:

.FSRPT	File Control Services work area and buffer pool vector
$OTSV	Language OTS work area vector
N.OVPT	Overlay Runtime System work area vector
$VEXT	Vector extension area pointer

The only proper reference to these pointers is by symbolic name.

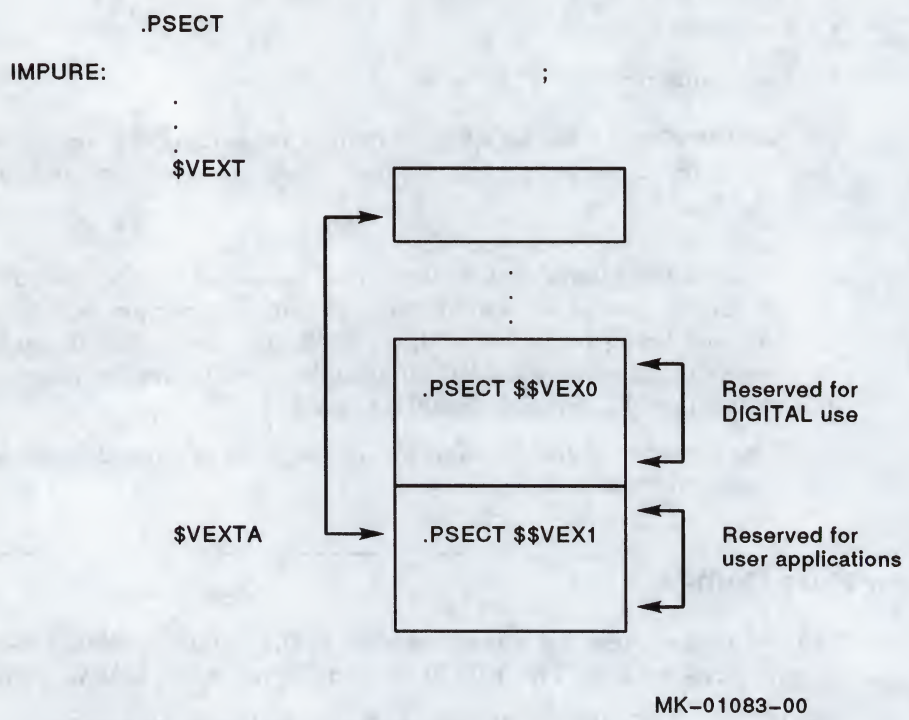
The Impure Area pointers contain the addresses of storage used by the reentrant library routines described above.

The address contained in the vector extension pointer locates an area of memory that can contain additional impure area pointers.

The format of the vector extension area is shown in Figure C–5. Each location within this region contains the address of an impure storage area that is referenced by subroutines that must be reentrant. Addresses below $VEXTA, referenced by negative offsets, are reserved for Digital applications. Addresses above this symbol, referenced by positive offsets, are allocated for user applications.

.PSECTs $$VEX0 and $$VEX1 have the attributes D, GBL, RW, REL, and OVR.

Figure C–5: Vector Extension Area Format



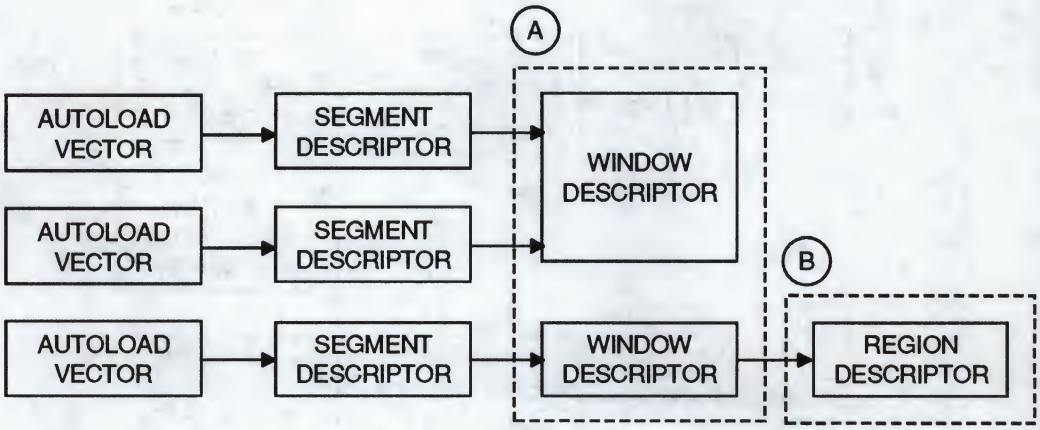
The .PSECT attribute OVR facilitates the definition of the offset to the vector and the initialization of the vector location at link time, as shown by the following example:

```
                .GLOBL      $VEXTA      ;MAKE SURE VECTOR AR A IS LINKED
                .PSECT      $$VEX1,D,GBL,RO,REL,OVR
BEG=.                       ; POINT TO BASE OF POINTER TABLE
                .BLKW        N          ; OFFSET TO CORRECT LOCATION
                                        ; IN VECTOR AREA
LABEL:          .WORD       IMPURE      ; SET IMPURE AREA ADDRESS
                                        ; DEFINE OFFSET
OFFSET==LABEL-BEG
                .PSECT
IMPURE:
                  .
                  .
                  .
```

C.3 Overlay Data Structure

Figure C–6 illustrates the structure and principal components of the task-resident overlay data base.

Figure C–6: Task-Resident Overlay Data Base

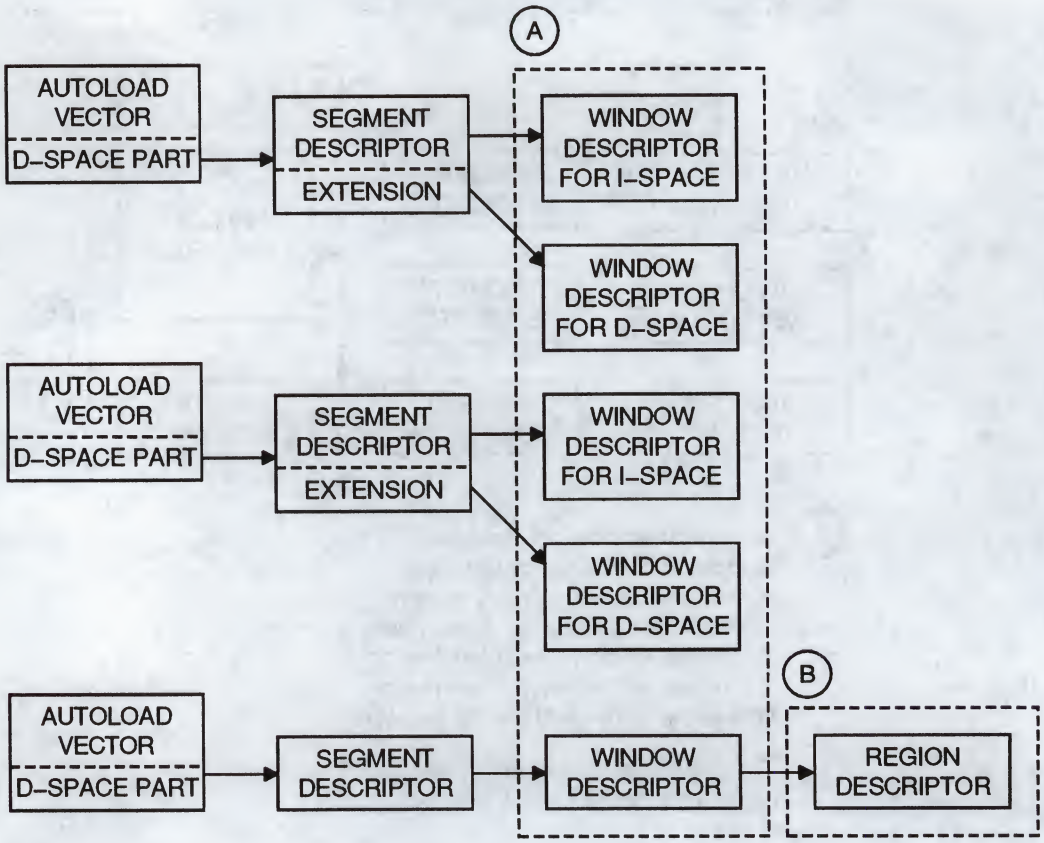


(A) Window descriptors are necessary for the windows that the overlay run–time system uses to map memory resident overlays. The overlay run–time system also needs window descriptors to map disk–resident overlays that are up–tree from memory–resident overlay segments.

(B) The overlay run–time system uses region descriptors to map overlaid libraries.

Figure C–7 illustrates the task-resident overlay database for an I- and D-space overlaid task.

Figure C–7: Task-Resident Overlay Database for and I- and D-Space Overlaid Task



(A) Window descriptors are necessary for the windows that the overlay run–time system uses to map memory resident overlays. The overlay run–time system also needs window descriptors to map disk–resident overlays that are up–tree from memory–resident overlay segments.

(B) The overlay run–time system uses region descriptors to map overlaid libraries.

Autoload vectors are generated whenever a reference is made to an autoloadable entry point in a segment located farther away from the root than the referencing segment.

One segment descriptor is generated for each overlay segment in the task or shared region. The segment descriptor contains information on the size, virtual address, and location of the segment within the task image file. In addition, it contains a set of link words that point to other segments. The overlay structure determines the link word contents.

The following sections describe the composition of each element.

C.3.1 Autoload Vectors for Conventional Tasks

The autoload vector table consists of one entry per autoload entry point in the form shown in Figure C–8.

Figure C–8: Autoload Vector Entry

Autoload Vector Entry

JSR @PC+2
Offset to pointer to autoload code
Segment descriptor address
Entry point address

MK–01055–00

The autoload vector contains a JSR to the autoload processor, $AUTO, followed by a pointer to the descriptor for the segment to be loaded and the real address of the entry point.

C.3.2 Segment Descriptor

The segment descriptor is composed of a six-word fixed-length portion. Segment descriptor contents are shown in Figure C–9.

Figure C–9: Segment Descriptor

TASK–RESIDENT SEGMENT DESCRIPTOR OFFSETS

15 – 12	11 – 0	BYTE
FLAGS	RELATIVE DISK BLOCK ADDRESS	0
VIRTUAL LOAD ADDRESS OF SEGMENT	F	2
LENGTH OF SEGMENT IN BYTES		4
LINK UP		6
LINK DOWN		10
LINK NEXT		12
SEGMENT NAME (2–WORD RADIX–50)		14
WINDOW DESCRIPTOR ADDRESS		20

FLAGS: 15–TASK RESIDENT FLAG (ALWAYS 1)
14–SEGMENT HAS DISK ALLOCATION (1=NO)
13–SEGMENT IS LOADED FROM DISK (1=YES)
12–SEGMENT IS LOADED AND MAPPED (0=YES)

F: 0–SEGMENT FOR I– AND D–SPACE TASK (1=YES)

TASK–RESIDENT SEGMENT DESCRIPTOR EXTENSION OFFSETS FOR I– AND D–SPACE TASKS ONLY

15 – 12	11 – 0	
UNUSED	D–SPACE DISK BLOCK ADDRESS	0
D–SPACE VIRTUAL LOAD ADDRESS		2
D–SPACE SEGMENT LENGTH IN BYTES*		4
D–SPACE WINDOW DESCRIPTOR ADDRESS		6

*0 IF ONLY I–SPACE SEGMENT

Word 0 contains the relative disk address in bits 0-11 and the segment status in bits 12-15. Each segment in the task image file begins on a disk block boundary. The relative disk address is the block number of the segment relative to the start of the root segment.

The segment flags are defined as follows:

Bit 15	Always set to 1.
Bit 14	0 = Segment loaded and mapped. 1 = Segment is either not loaded or not mapped.
Bit 13	0 = Segment has disk allocation. 1 = Segment does not have disk allocation.
Bit 12	0 = Segment not loaded from disk. 1 = Segment loaded from disk.

Word 1 contains the load address of the segment. This address is the first virtual address of the area where the segment will be loaded.

Word 2 specifies the length of the segment in bytes.

The next three words point to the following segment descriptor:

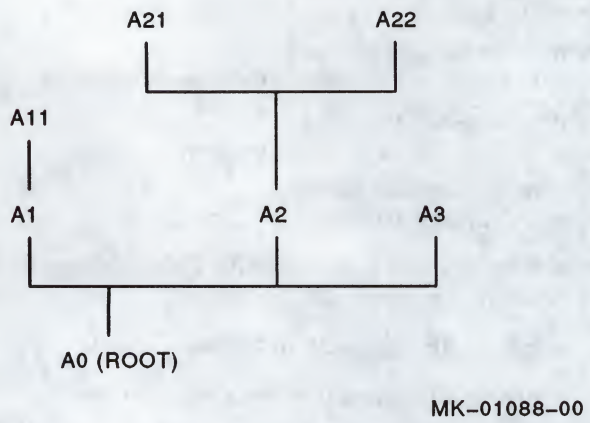
Link Up	Points to the next segment away from the root. Link Up equals 0 of you are already at the leaf.
Link Down	Points to the next segment toward the root. Link Down equals 0 if you are already at the root.
Link Next	Points to the adjoining segment. Link Next equals the address of the current segment if there are no others on the same level with the same Link Down. Link Next links all segments on the same level that have the same Link Down in a circular fashion. Thus, in Figure C–10, Link Next in A3 points to A1, but Link Next in A11 points to A11 itself and Link Next in A0 points to A0 itself.

The segment descriptor for an I- and D-space task consists of a fixed part that is nine words long and an optional part that is four words long. The optional part is always present for task segments and never present for library segments. The bottom half of Figure C–9 illustrates the contents of the segment descriptor for I- and D-space tasks.

When a segment is loaded, the overlay run-time system follows the links to determine which segments are being overlaid and should therefore be marked out of memory.

Using the tree in Figure C–10 as an example:

Figure C–10: Sample Tree



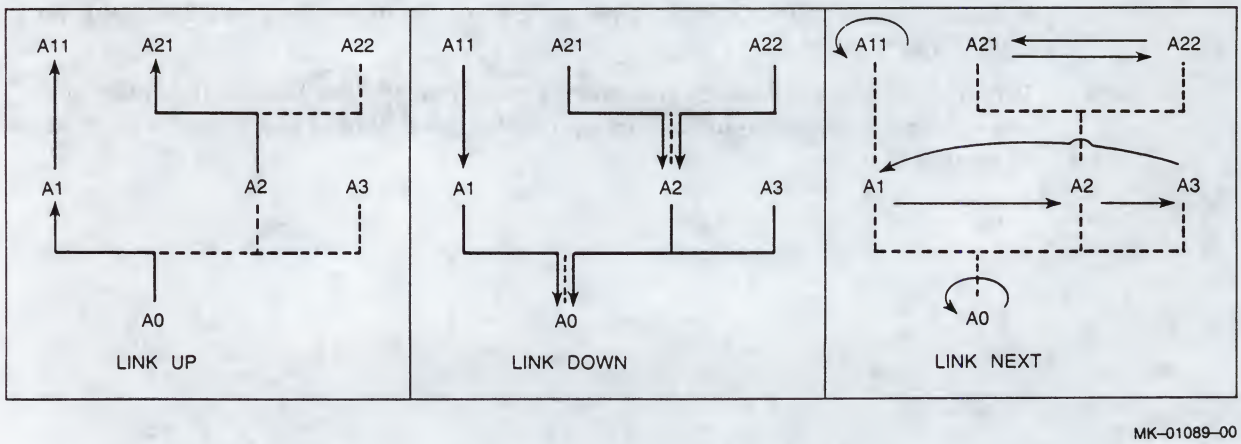
The segment descriptors are linked as in Figure C–11.

If there is a co-tree, the Link Next for the root segment descriptor points to the co-tree root segment descriptor.

Words 6 and 7 contain the segment name in Radix-50 format.

Word 8 points to the window descriptor used to map the segment (0 = none).

Figure C–11: Segment Linkage Directives



C.3.2.1 Autoload Vectors for I- and D-Space Tasks

The autoload vector table consists of two entries (put into the task image for each autoload entry point) in the form shown in Figure C-12.

The I-space part of the autoload vector contains a move (MOV) instruction that places the address of the D-space part of the vector on the stack. The vector then executes an indirect jump (JMP) to $AUTO through .NAUTO. The D-space part of the vector contains the segment descriptor address of the required routine.

Figure C–12: Autoload Vector Entry for I- and D-Space Tasks

MOV (PC) +, – (SP)
ADDRESS OF PACKET (D–SPACE)
JMP @.NAUTO
PC RELATIVE OFFSET TO .NAUTO

I–SPACE PORTION

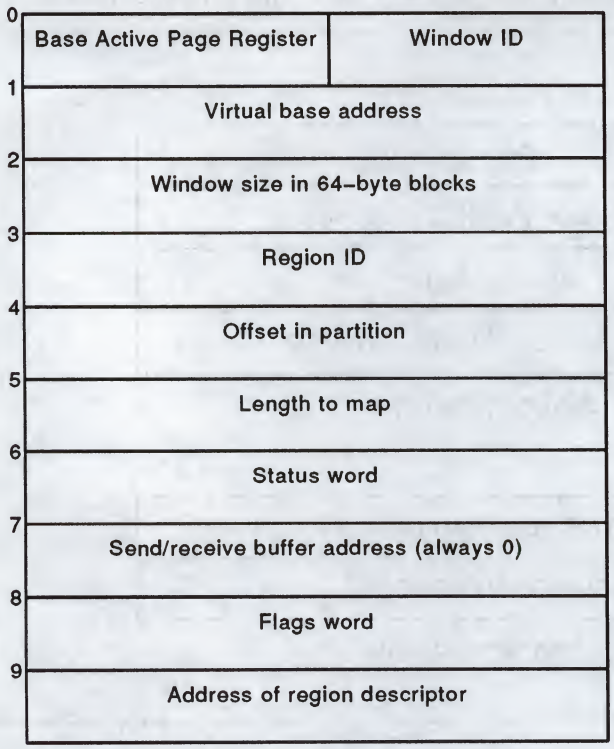
ADDRESS OF SEGMENT DESCRIPTOR
ENTRY POINT ADDRESS

D–SPACE PORTION

C.3.3 Window Descriptor

TKB allocates the window descriptors only if you define a structure containing memory-resident overlays. Figure C–13 illustrates the format of a window descriptor.

Figure C–13: Window Descriptor



MK–01087–00

Words 0 through 7 constitute a window descriptor in the format required by the mapping directives (the Program Logical Address Space (.PLAS) Mapping Directives - see the *RSTS/E System Directives Manual* for more information). The overlay loading routine fills in the region ID at run time.

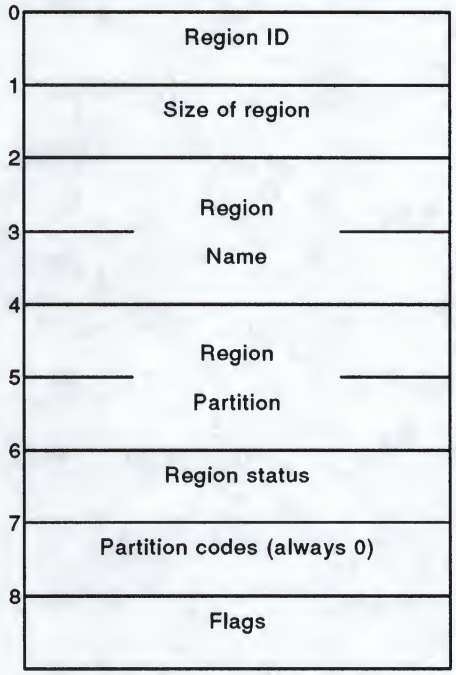
Words 8 and 9 contain additional data that the overlay routines refer to. Bit 15 of the flags word, if set, indicates that the window is currently mapped into the task's address space.

Word 9 contains the address of the associated region descriptor.

C.3.4 Region Descriptor

Figure C–14 illustrates the format of a region descriptor.

Figure C–14: Region Descriptor



MK–01086–00

Words 0 through 7 constitute a region descriptor in the format required by the mapping directives. Word 8, the flags word, is referred to by the overlay load routine. Bit 15 of the flags word, if set, indicates that a valid region identification is in word 0.

C.4 Root Segment

The root segment is written as a contiguous group of blocks. The root segment is the first segment loaded and remains in memory for the entire life of the program execution.

C.5 Overlay Segments

Each overlay segment begins on a block boundary. The relative block number for the segment is placed in the segment table. Note that a given overlay segment occupies as many contiguous disk blocks as it needs to supply its space request. The maximum size for any nonroot segment is 28K words. The maximum size for the root segment is 32K words.

Appendix D

Reserved Symbols

All symbols and PSECT† names containing a period (.) or dollar sign ($) are reserved for Digital-supplied software. Several global symbols and PSECT† names are reserved for use by the Task Builder. Special handling occurs when a definition of one of these names is encountered in a task image.

The definition of a reserved global symbol in the root segment causes a word in the task image to be modified with a value calculated by the Task Builder. The relocated value of the symbol is taken as the modification address.

Table D–1 shows global symbols reserved by the Task Builder.

Table D–1: Task Builder Reserved Global Symbols

Global Symbol	Modification Value
$ALERR	OTS address of overlay load eroor handler.
$AUTO	OTS address of autoload routine.
$DBTS	Debugger time stamp.
.FSRPT	Address of file storage region work area (.FSRCB).
$FSTIN	OTS address of fast map overlay routine.
$MARKS	OTS address of MARK segment routine.
.MBLUN	Mailbox logical unit number.
.MOLUN	Error message output device.
.NALER	OTS entry point to overlay load error handler.
.NAUTO	OTS entry point to $AUTO or $LOAD.
.NDTDS	OTS highest displaced segment.
.NFAST	OTS AST supression control flags.
.NFMAP	OTS entry point to Fast Map initialization routine.
.NIOST	OTS common I/O status doubleword.
.NLUNS	The number of logical units used by the task, not including the message output and overlay units.
.NMARKS	OTS entry point to MARK segments.

(continued on next page)

† PSECTS are created by .ASECT, .CSECT, or .PSECT directives. The .PSECT directive eliminates the need for either the .ASECT or .CSECT directive, both of which are retained only for compatibility with other systems. In this document all sections are referred to as PSECTS unless the specific characteristics of .ASECT or .CSECT apply.

Table D–1 (Cont.): Task Builder Reserved Global Symbols

Global Symbol	Modification Value
.NOVLY	The overlay logical unit number.
N.OVPT	Address of overlay run-time system work area (.NOVLY).
.NRDSG	OTS entry point to READ segments.
.NSTBL	The address of the segment description tables. This location is modified only when the number of segments is greater than one.
.NSZSG	OTS size of resident segment descriptors.
.ODTL1	Logical unit number for the ODT terminal device TI:.
.ODTL2	Logical unit number for the ODT line printer device CL:.
$OTSV	Address of Object Time System work area ($OTSVA).
.PTLUN	Logical unit number for plotter/graphics software.
$RDSEG	OTS address of READ segment routine.
.SUML1	P/OS standard utility module LUN.
.TRLUN	The trace subroutine output logical unit number.
.USLU1	Logical unit number for special purpose user software.
.USLU2	Logical unit number for special purpose user software.
$VEXT	Address of vector extension area ($VEXTA).

The following global symbols are reserved by TKB for tasks using disk-resident overlays:

Global Symbol	Modification Value
$MARDS	OTS entry point to I/D MARK segment routine.
$MAFKS	OTS entry point to optimized MARK segment routine.
$MAFDS	OTS entry point to optimized I/D MARK segment routine.

The following global symbols are reserved by TKB for tasks using memory-resident overlays:

Global Symbol	Modification Value
$MARKR	OTS entry point to MARK segment routine.
$MARDR	OTS entry point to I/D MARK segment routine.
$$MAFKR	OTS entry point to optimized MARK segment routine.
$MAFDR	OTS entry point to opitmized I/D MARK segment routine.

The following global symbols are reserved by TKB for tasks using cluster libraries:

Global Symbol	Modification Value
$MARKC	OTS entry point to MARK segment routine.
$MARDC	OTS entry point to I/D MARK segment routine.

Global Symbol	Modification Value
$MAFKC	OTS entry point to optimized MARK segment routine.
$MAFDC	OTS entry point ot optimized I/D MARK segment routine.

The PSECT names in Table D–2 are reserved by the Task Builder. In some cases, the definition of a reserved PSECT causes the PSECT to be extended if the appropriate option is specified.

Table D–2: PSECT Names Reserved by the Task Builder

Source Location	Section Name	Description
TKB	$$ALER	Contains code to process or trap Overlay Run-time system segment load errors. Provides named areas in the task for the FORTRAN Object Time System and the RSX Overlay Run-time System.
TKB	$$ALVC	Contains the segment autoload vectors for tasks without I- and D-space.
TKB	$$ALVD	Contains the D-space portions of the segment autoload vectors in an I- and D-space task.
TKB	$$ALVI	Contains the I-space portions of the segment autoload vectors in an I- and D-space task.
TKB	$$AUTO	Contains code to determine if a called subroutine in an overlay segment is already in memory or if that overlay segment should be read into memory before control is passed to the subroutine that is called.
Input Module	$$DBTS	This symbol should appear in the debugger input module with the symbol $DBTS as follows: .PSECT $$DBTS .$DBTS:: .PSECT The task builder extends $$DBTS and fills it with time stamp information followed by the filename information of the .STB file.
SYSLIB	$$DEVT	The extension length (in bytes) is calculated from the formula: EXT = (S.FDB+52)*UNITS The definition of S.FDB is obtained from the root segment symbol table, and UNITS is the number of logical units used by the task, excluding the message output, overlay, and ODT units.
SYSLIB	$$FSR1	The extension of this section is specified by the ACTFIL option.
SYSLIB	$$FTSM	Contains the code to map memory-resident overlays using the fast map facility instead of the standard executive mapping directive CRAW$.
SYSLIB	$$IOB1	The extension of this section is specified by the MAXBUF option.

(continued on next page)

Table D–2 (Cont.): PSECT Names Reserved by the Task Builder

Source Location	Section Name	Description
TKB	$$IOB2	A zero length .PSECT containing a label, IOBFND, that is stored in the work area offset, W.BEND, representing the upper bound of the I/O buffer, $$IOB1. TKB uses $$IOB2 as a boundary value to determine whether the I/O buffer has overflowed.
TKB	$$LOAD	Overlay manual load routine.
TKB	$$MRKS	Contains code to properly mark those segments that are not needed any longer or have been overlaid by another segment as being out of memory. This ensures that a fresh copy of the overlay segment will be read in the next time the overlay segment is needed.
SYSLIB	$$OBF1	FORTRAN OTS uses this area to parse array type format specifications. This section can be extended by the FMTBUF keyword.
TKB	$$OBF2	A zero length .PSECT containing a label, OBFH, that is stored in the work area offset, W.OBFH, which represents the upper bound of the run-time format buffer, $$OBF1. TKB uses $$OBF2 to determine if the run-time format buffer has overflowed.
TKB	$$OVDT	The Overlay Run-time System impure data area. The symbol N.OVPT in low memory points to this area. This area defines the operational parameters with which the Overlay Run-time system operates on disk-resident and memory-resident overlay structures.
TKB	$$OVRS	The .ABS. program section that redefines the Overlay Run-time System impure data area with different symbols, defined as offsets and relative to zero. These offsets are necessary for proper linkages between the subroutines in the Overlay Run-time System. This program section is never included in the memory allocation of the task because of its absolute program section attribute.
TKB	$$PDLS	Cluster library service routine.
TKB	$$RDSG	Contains the code that reads into memory the overlay segment selected by the code contained in the programs section $$AUTO.
TKB	$$RGDS	Contains the region descriptors for resident libraries referred to by the task.
TKB	$$RTQ	Defines the PSECT used for selective enabling of AST recognition in the Overlay Run-time System. $$RTQ is 0 in length if $AUTOT is not included.
TKB	$$RTR	Defines the PSECT used for selective disabling of AST recognition in the Overlay Run-time System. $$RTR is 0 in length if $AUTOT is not included.
TKB	$$RTS	Contains the return instruction.

(continued on next page)

Table D–2 (Cont.): PSECT Names Reserved by the Task Builder

Source Location	Section Name	Description
TKB	$$SLVC	Supervisor-mode library transfer vectors (RSX-11M-PLUS only).
TKB	$$SGD0	Contains the program section adjoining the task segment descriptors.
TKB	$$SGD1	Contains the task segment descriptors.
TKB	$$SGD2	Contains a .WORD 0 following the task segment descriptors.
FORTRAN-77	$$TSKP	TKB fills in the following words in the PSECT (note that the word values are filled into the Section in order): • APR bit map in word $APRMP • Task offset into region in word $LBOFF • Maximum physical read/write memory needed for task in word $MXLGH • Maximum physical read-only memory needed for task in word $MXLGH+2 • Task extension in 32-word blocks in word BOK$LBEXT • Total contribution of FORTRAN virtual arrays • Maximum physical read-only D-space memory needed for task in word $MXLGH+4 • Maximum physical read/write D-space memory needed for task in word $MXLGH+6
MACRO-11	$$TSKP	TKB fills in the first work of the PSECT called TSKP$. Refer to the *System Directives Manual*for more details.
TKB	$$WNDS	Contains task window descriptors.

Appendix E

Improving Task Builder Performance

This appendix contains procedures and suggestions to help you maximize Task Builder performance. Procedures are given for:

- Evaluating and improving Task Builder throughput
- Modifying command switch defaults to provide a more efficient user interface

E.1 Evaluating and Improving Task Builder Performance

Task Builder throughput is determined by these factors:

- The amount of memory available for table storage
- The amount of disk latency due to input file processing

The discussion in the following paragraphs outlines methods for improving throughput in each case. The methods approach their goals through judicious use of system resources and Task Builder features.

E.1.1 The Task Builder Work File

The largest factor affecting Task Builder performance is the amount of memory available for table storage. To reduce memory requirements, the Task Builder uses a work file to store symbol definitions and other tables. If the total size of these tables is within the limits of available memory, the work file is kept in core and not shunted to a disk. If the tables exceed the amount of memory available, some information must be moved to the disk, which degrades performance.

Work file performance can be gauged by consulting the statistics portion of the Task Builder map. The following parameters are displayed:

Number of work file references:

Total number of times that work file data was referenced.

Work file reads:

Number of work file references that resulted in disk accesses to read work file data.

NOTE

If work file reads and writes equal zero and the number of work file references is greater than zero, you can be sure that the work file remained in memory.

Work file writes:

Number of work file references that resulted in disk accesses to write work file data.

Size of Core Pool:

Amount of in-core table storage in words. This value is also expressed in units of 256-word pages (information is read from and written to disk in blocks of 256 words.)

Size of Work File:

Amount of work file storage in words. If this value is less than the core pool size, the number of work file reads and writes is zero. That is, no work file pages are removed to the disk. This value is also expressed in pages (256-word blocks).

Elapsed Time:

Amount of time required to build the task image and produce the map. This value excludes ODL processing, option processing, and the time required to produce the global cross-reference.

The overhead for accessing the work file can be reduced in one or more of the following ways:

- By increasing the amount of memory available for table storage
- By placing the work file on the fastest random access device, such as the virtual disk (DV:)
- By decreasing system overhead required to access the file
- By reducing the number of work file references

The Task Builder automatically increases its size up to the maximum job size, which may be as large as 32K words. See the *RSTS/E System Manager's Guide* for information on how to change the maximum job size.

The size of the work file can be reduced by:

- Linking your task to a core-resident run-time system containing commonly used routines (for example, BASIC-PLUS-2 object time system) whenever possible
- Including common modules, such as components of an object time system, in the root segment of an overlaid task
- Using an object library file of concatenated object modules if many modules are to be linked

In the last two cases, system overhead is also significantly reduced because fewer files must be opened to process the same number of modules.

The number of work file references can be reduced by eliminating unneeded output files and cross-reference processing or by obtaining the short map. In addition, selected files, such as the default system object module library, can usually be excluded from the map. In this case, a full map can be obtained at less frequent intervals and retained.

Try the following procedures to improve work file performance:

- Install I- and D version of TKB.
- Decrease work file size by using resident run-time systems, concatenated object files, and object libraries.
- Decrease work file size by moving common modules into the root segment of an overlaid task.
- Decrease the number of work file references by eliminating the map and global cross-reference, obtaining the short map, or excluding files from the map.
- Place the work file on the fastest possible device. If the system manager installs a system-wide logical "device:OV", the Task Builder uses a device other than SY: as the work file device.

  If the device is a private pack, all accounts of any user wishing to use the Task Builder must be entered on the private pack while the system-wide logical is in effect. Otherwise, a protection violation error occurs for those users without accounts when the Task Builder tries to create its work file.

  Again, make sure the device is mounted so users without access privileges will not obtain fatal errors when the Task Builder tries to create its work file.
- Use the CCL/SI:## to increase size to maximum immediately. This may reduce swapping when TKB must increase in size.

E.1.2 Input File Processing

The suggestions for minimizing the size of the work file and number of work file accesses also drastically reduce the amount of input file processing.

A given module can be read up to three times when building the task:

1. To build the symbol table
2. To produce the task image
3. To produce the long map

Files that are excluded from the long map are read only twice. The third pass is completely eliminated for all modules when a short map is requested. So, if you do not need the long map, use the /SH switch (described in Section 11.23) to eliminate the third pass.

Index

A

ABORT option, 12–3
ABS attribute, 13–4
Absolute Patch (ABSPAT), 12–4
Absolute Patch for D-Space(DSPPAT), 12–11
Absolute resident area, 7–2, 7–3
ABSPAT option, 12–4
Access code (resident library), 2–11
Access code in CLSTR option, 12–8
Access code in cluster libraries, 2–14
Access Resident Common Block (RESCOM), 12–26
Access Resident Library (RESLIB), 12–27
Access System Common Block (COMMON), 12–10
Access System-Owned Resident Library (LIBR), 12–21
ACTFIL option, 12–5
Active files, 12–5
Active Page Register, 2–11, 12–8, 12–21
Additive relocation, B–26
Addresses
  absolute, 7–2
  relative, 7–2
Address space, 2–2 to 2–3
$$ALVC, 6–7
Ambiguously defined symbols
  in a simple overlay, 3–12
  in co-trees, 4–6
APR, 2–2, 2–11, 12–8, 12–21
  with cluster libraries, 2–14
  with I- and D-space tasks, 8–1
Area, memory-resident, 7–1 to 7–7, 12–25
ASECT, B–5
ASG option, 12–6
  example, 2–15
Assembler (MAC), 2–9
  used with TKB, 1–1
Assembly language and cluster libraries, 7–9
Assign Devices (ASG), 12–6
Asterisk (*)
  before .FCTR names, 5–5
  before .NAME names, 5–5
  before file names, 5–4
  before items in parentheses, 5–5
  before program sections, 5–4
  easiest use of, 4–3, 5–1, 13–5
  errors in using, 5–6
  for co-trees, 4–3
  for simple overlays, 3–4
  not for null co-tree roots, 4–5
  ODL operator, 13–5
Attributes, 6–1 to 6–4, 13–3 to 13–4
Attributes (Cont.)
  CON, 12–12
  OVR, 12–12
$AUTO, 5–2, 7–9
$$AUTO, 6–7
Autoloadable library entry point item type, B–30
Autoload indicator, 5–1 to 5–6, 13–5
  for co-trees, 4–3
  for simple overlays, 3–4
  not for null co-tree roots, 4–5
Autoloading a data PSECT, 6–5
Autoload processor, 5–2
Autoload routines ($AUTO), 7–9
Autoload vector, 3–4, 13–5, B–30, C–13
  definition, 5–1
  how to request specific, 5–4
  specific examples, 5–6
  where needed, 5–3

B

.B2S file, 4–8
BASIC Object Time System, 2–7
BASIC-PLUS-2, 1–1
  disk libraries for, 2–4t
  example build, 2–15
  libraries in a cluster, 12–7
  resident libraries for, 2–7
  run-time system for, 2–2
Blank common area, 6–7
BLDODL utility, 3–2
.BLK, 6–7
BP2OTS.OLB, 2–4t, 2–7
BP2RES library, 2–7, 2–12, 12–7
BP2SML library, 2–7, 2–12, 12–7
Branch (overlay structure), 3–10
Buffer
  format, 12–14
  record, 12–23
Build a Common Block Shared Region (/CO), 11–4
Build a Library Shared Region (/LI), 11–16

C

C81CIS.OLB, 2–5t
C81CIS library, 2–12, 12–7
C81LIB.OLB, 2–5t
C81LIB library, 2–12, 12–7
Calls
  between cluster libraries, 7–10
  cross-tree, 4–3

Calls (Cont.)
logical independence of, 3–2, 3–8, 3–10
Call structure, 3–2, 4–1
with co-trees, 4–7
CCL command, 2–7
/CC switch, 11–3
Characters within the SYSTAT program name, 12–33
CIS option, 2–5t
CLSTR option, 2–11, 12–6
format, 2–13
Cluster libraries, 2–12f
and assembly language, 7–9
and calls between libraries, 7–10
and memory-resident overlays, 7–8
and the CLSTR option, 2–11, 12–6
Building, 7–8
GBLINC option, 12–16
GBLXCL option, 12–19
limitations of use, 2–14
revectoring, 7–10
trapping or asynchronous entry, 7–9
Cluster Libraries (CLSTR), 12–6
.CMD files, 10–4
CMPRT option, 12–9
COBLIB.OLB, 2–4t
COBOL (PDP-11), 1–1
disk libraries for, 2–4t
example build, 2–16
run-time system for, 2–2
COBOL-81, 1–1
disk libraries for, 2–5t
example build, 2–16
libraries in a cluster, 12–7
run-time system for, 2–2
symbolic debugger, 2–8
COBOVR.OLB, 2–5t
Code
sharable, 2–7
Comma
ODL operator, 3–4, 13–4
ODL operator (co-trees), 4–3
Command
CCL, 2–7
multiline, 2–9, 10–3
Command line
ending TKB, 10–3
ODL, 13–1
TKB, 2–8, 10–1
Comments, 10–6
Commercial instruction set option, 2–5t
Common area
allocating space for, 6–2
blank, 6–7
definition, 6–2
resident, 7–1, 12–26
COMMON option, 12–10
Comparison of disk and resident libraries, 2–7
Compilers, used with TKB, 1–1
Compiling (BASIC-PLUS-2 sample), 4–8
Completion Routine (CMPRT), 12–9
Complex relocation, B–24
CON attribute, 6–4, 12–12, 13–4
Concatenated programs and subprograms (/CC), 11–3
Concatenation, 3–4, 3–13, 13–4
Concurrent libraries, 8–4
Context, low core, C–9
Control section, B–5
Core common, 11–12
/CO switch, 11–4
Co-trees, 4–1 to 4–17, 11–11
and high-level languages, 4–6 to 4–17
fine-tuning, 4–13
how loaded during execution, 4–3, 4–3f
most space-saving, 4–5 to 4–6
sample program, 4–6
structure, 4–1
Cross-tree calls, 4–3
CSECT, B–5

D

DAPRES library, 2–12
Dash
See *Hyphen*
/DA switch, 11–5, B–26
Data PSECT, 6–5
Data space
See *I- and D-space tasks*
D attribute, 6–4, 6–5, 13–4
DBLLIB.OLB, 2–4t
DBRLIB.OLB, 2–4t
DBRRES, 2–4t
$$DBTS, B–29
DCL (LINK command), 1–4
Debugger, B–31
Debugger (COBOL-81), 2–8
Debugging Aid (/DA), 11–5, B–26
Declare Stack Size (STACK), 12–31
Default library, 3–15, 11–6, 11–11, 11–17
how searched for co-trees, 4–3
in CLSTR option, 2–13, 12–6
using co-tree techniques on, 4–17
Default Library (/DL), 11–6
Define a Global Symbol (GBLDEF), 12–15
Define High Segment (HISEG), 12–20
Device
assigning, 12–6
Device designators, specifying, 2–9
Diagnostic
errors, A–1
messages, omitting, 11–20
run, 10–2
DIBOL, 1–1
disk libraries for, 2–4t
example build, 2–16
resident libraries for, 2–4t
run-time system for, 2–2
DIBOL library, 2–12, 12–7
DIBOL Management System, 2–4t
Directive emulation code for RSX, 2–11
Directory, internal symbol, B–26
Disappearing RSX run-time system, 2–11
Disk access time, reducing, 3–9 to 3–10
Disk and resident libraries, comparison of, 2–7
Disk libraries, 2–4 to 2–5, 2–5f, 2–8, 11–14
/DL switch, 11–6
DMS, 2–4t
Double slash (//), to end TKB, 10–3
DSK attribute, 13–3
DSPPAT option, 12–11
Dump, 11–22

E

/EL switch, 11–7
.END command, 3–3 to 3–5, 13–2
End-of-module record, B–36
Enter Options prompt, 10–3
Error messages, A–1 to A–8
  diagnostic, A–1
  fatal, A–1
Exclamation point (!), 7–5, 11–23
  ODL operator, 13–4
Exclude Global from .STB File (GBLXCL), 12–19
Executable program
  extending size of, 12–13
  file, 2–8
  file format, C–1f
  patching, 12–4
Executable program file, 7–2
Exit on Error (/XT), 11–38
Extend Library, 11–7
Extend Program Section (EXTSCT), 12–12
Extend Task Memory (EXTTSK), 12–13
EXTSCT option, 12–12
EXTTSK option, 12–13
  example, 2–15

F

F4PCLS library, 2–12, 12–7
F4POTS.OLB, 2–5t
F4PRMS.OLB, 2–5t
Fast Map, 11–8
Fast Map overlay (/FO), 11–9
Fast-mapping code, 7–29
Fast-Mapping Facility, 7–28 to 7–31
Fatal errors, A–1
FCS, 11–33
.FCTR command, 3–3 to 3–5, 13–2
  nesting limit, 3–5
FDVDBG.OLB, 2–5t
FDVLIB.OLB, 2–5t
FDVRDB library, 2–12, 12–7
FDVRES library, 2–12, 12–7
File
  Control System, 11–33
  declaring maximum open, 12–5
  executable, 2–8, 7–2, C–1
  indirect command, 10–4 to 10–5
  input to TKB, 10–2
  library, 11–14
  map, 2–8, 10–1
  memory map, 11–17
  object, 2–4, 2–8
  specifications, 10–6
  symbol table, 2–8, 7–2, 10–1, 12–20
  task, 2–8, 7–2, 10–1
FIRQB, 11–12
Floating-point processor, 11–10
FMS
  disk libraries for, 2–5t
  libraries in a cluster, 12–7
/FM switch, 11–8
FMTBUF option, 12–14
Format Buffer Size (FMTBUF), 12–14
FORTRAN-77, 1–1
  disk libraries for, 2–5t
  example build, 2–17
FORTRAN-77 (Cont.)
  resident library for, 2–12, 12–7
  run-time system for, 2–2
FORTRAN virtual arrays, 7–13
/FO switch, 11–9
/FP switch, 11–10
.FSRPT, C–9
/FU switch, 4–17, 11–11
  full search, 11–11

G

GBL attribute, 6–5, 13–4
  segment name, 13–3
GBLDEF option, 12–15
GBLINC option, 7–12, 12–16
  in cluster library example, 7–12
GBLPAT option, 12–17
GBLREF option, 7–6, 12–18
GBLXCL option, 7–12, 12–19
Global additive displaced relocation, B–18
Global additive relocation, B–18
Global displaced relocation, B–17
Global Relative Patch (GBLPAT), 12–17
Global relocation, B–16
Global symbol item type, B–32
Global Symbol Reference (GBLREF), 12–18
Global symbols
  ambiguously defined, 3–12, 4–6
  autoload vectors for, 5–2
  defining, 12–15
  definition, 3–11
  excluding from .STB file, 12–19
  forcing reference in root, 7–6
  general discussion, 1–3
  including in .STB file, 12–16
  in internal file, 11–34
  multiply defined, 3–12, 4–6
  name entry, B–7
  reference from root, 12–18
  reserved, D–1
  undefined, 3–12, 4–6
GSD, B–1 to B–4, B–6, B–9, B–12

H

/HD switch, 11–12
Header, 11–12, C–6 to C–9
High segment, 1–3, 12–20
HISEG option, 12–20
Hyphen, 3–13
  in .ROOT and .FCTR commands, 3–4 to 3–5
  ODL operator, 3–4, 13–4
  with library files, 3–5

I

I-and D-Space (/ID), 11–13
I- and D-space tasks, 8–1 to 8–4
I attribute, 6–4, 13–4
/ID switch, 11–13
Impure area, C–9
Include Global in .STB File (GBLINC), 12–16
Indirect command files
  ODL, 13–5
  TKB, 10–4 to 10–5
Input files, 10–2

Instruction space
See *I- and D-space tasks*
Internal
displaced relocation, B–16
relocation, B–15
symbol directory, B–26
symbol name, B–6
Internal symbol name item type, B–34
$$IOB1, 12–23
ISD record, B–1
description, B–26
general format, B–27
types, B–26

J

Job area, 2–2 to 2–3
JSR PC instruction, 7–9
Jump table, 7–10

L

L$BBLK, C–5
L$BDAT, C–4
L$BDHV, C–5
L$BDLZ, C–5
L$BDMV, C–5
L$BDMZ, C–5
L$BEXT, C–4
L$BFL2, C–5
L$BFLG, C–4
L$BHDB, C–5
L$BHGV, C–4
L$BHRB, C–5
L$BLDZ, C–4
L$BLIB, C–4
L$BLUN, C–5
L$BMXV, C–4
L$BMXZ, C–4
L$BOFF, C–4
L$BPAR, C–4
L$BPRI, C–4
L$BRDL, C–5
L$BROB, C–5
L$BROL, C–5
L$BSA, C–4
L$BSEG, C–4
L$BSGL, C–4
L$BSYS, C–4
L$BTSK, C–4
L$BWND, C–4
L$BXFR, C–4
Label block group, C–2 to C–4
Languages, used with TKB, 1–1
LB:, 2–10 to 2–11
LBR utility, 11–36
/LB switch, 2–8, 3–5, 11–14
naming specific routines, 3–15, 4–16, 11–14
LCL attribute, 13–4
LD$ACC, C–6
LD$CLS, C–6
LD$REL, C–6
LD$RSV, C–6
LD$SUP, C–6
Libraries, 1–2, 2–1 to 2–17
BP2RES, 2–12, 12–7
Libraries (Cont.)
BP2SML, 2–12, 12–7
C81CIS, 2–12, 12–7
C81LIB, 2–12, 12–7
clustering resident, 2–11, 2–12f
DAPRES, 2–12
default, 3–15
default in CLSTR option, 12–6
DIBOLR, 2–12, 12–7
disk, 2–4 to 2–5, 2–5f, 2–8
F4PCLS, 2–12, 12–7
FDVRDB, 2–12, 12–7
FDVRES, 2–12, 12–7
indicating in ODL files, 3–13
object, 2–4
resident, 2–5 to 2–7, 2–10, 7–1, 12–21, 12–27
RMSRES, 2–12, 12–7
routines inserted in co-trees, 4–6
routines inserted in overlays, 3–13
rules for building cluster, 7–8
SMRES, 2–12, 12–7
Library account (LB:), 2–10
Library File (/LB), 11–14
LIBR option, 2–10 to 2–11, 12–21
example, 2–17
.LIMIT (MACRO directive), B–20
Line-number or PC correlation item type, B–34
LINK command, 1–4
Linking, general discussion, 1–2 to 1–3
/LI switch, 11–16
Literal record type, B–36
Local symbols, 3–11
Location counter, B–19 to B–20
Logical independence, 3–2, 3–8, 3–10
Logical units
assigning, 12–6
declaring maximum number of, 12–35
Low core context, C–9

M

MAC assembler, 1–1, 2–9
MACRO programs, 2–17
run-time system for, 2–2
with I- and D-space tasks, 8–1 to 8–4
MAKSIL, 7–1, 11–19
Map, 6–6
132 columns, 11–37
80 columns, 11–37
detailed description, 11–26 to 11–31
file, 3–6, 6–6, 10–1, 11–17
first 1000 bytes in, 6–7
long, 11–26
overlay description, 3–7e
sample, 4–9e, 6–9 to 6–15
sample with co-trees, 4–11e, 4–16
short, 11–26
spooling, 11–32
Map Contents of File (/MA), 11–17
.MAP file, 2–8
Mapping, 2–5, 2–11
Map Supervisor D-Space, 9–22 to 9–24
/MA switch, 4–17, 6–6, 11–17
MAXBUF option, 12–23
Maximum Number of Units or Channels (UNITS), 12–35
Maximum Record Buffer Size (MAXBUF), 12–23

Maximum size, 2–2
Memory map, 3–6, 6–6
  132 columns, 11–37
  80 columns, 11–37
  detailed description, 11–26 to 11–31
  file, 10–1, 11–17
  long, 11–26
  overlay description, 3–7e
  sample listing, 6–9 to 6–15
  short, 11–26
  spooling, 11–32
Memory-resident overlays, 7–4 to 7–7, 11–23
Memory-resident overlays in cluster libraries, 7–8
Mode-Switching Vectors, 9–1
Module, B–1
  end-of-module record, B–36
  general discussion, 1–2 to 1–3
  general format, B–1f
  name, B–5
Module name item type, B–31
/MP switch, 3–6, 10–2, 11–18
MRG utility, 3–2
MSDS$ call, 9–23
Multiline command, 2–9, 10–3
Multiple builds in one run, 10–4
Multiply defined symbols
  in a simple overlay, 3–12
  in co-trees, 4–6
Multiuser Program (/MU), 11–19
/MU switch, 11–19

N

N.OVPT, C–9
.NAME
  for null co-tree root, 4–4
  to make data PSECT autoloadable, 6–5
.NAME command, 13–2
Nested .FCTR commands, 3–5, 13–2
Nested parentheses, 3–9, 13–5
/NM switch, 11–20
No Diagnostic Messages (/NM), 11–20
NODSK attribute, 13–3
NOGBL attribute, 13–3
Null root for co-tree, 4–4
Number of Active Files (ACTFIL), 12–5
Number of Address Windows (WNDWS), 12–39

O

$$OBF1, 12–14
Object files, 2–8
Object library file type, 2–4
.OBJ file, 2–8, 10–2, B–1
  general format, B–1f
ODL, 13–1 to 13–5
  command line, 13–1
ODL file, 3–1 to 3–6, 10–2, 13–1
ODL operators, 3–4, 4–3, 4–5, 5–4 to 5–6, 13–4
ODT, 11–5, 12–24
ODT SST Vector (ODTV), 12–24
ODTV option, 12–24
.OLB file, 2–4, 2–8, 10–2
Option
  ABORT, 12–3
  ABSPAT, 12–4
  ACTFIL, 12–5
Option (Cont.)
  ASG, 12–6
  CLSTR, 2–11, 12–6
  CMPRT, 12–9
  COMMON, 12–10
  DSPPAT, 12–11
  EXTSCT, 12–12
  EXTTSK, 12–13
  FMTBUF, 12–14
  GBLDEF, 12–15
  GBLINC, 7–12, 12–16
  GBLPAT, 12–17
  GBLREF, 12–18
  GBLXCL, 7–12, 12–19
  HISEG, 12–20
  LIBR, 12–21
  MAXBUF, 12–23
  ODTV, 12–24
  PAR, 12–25
  RESCOM, 12–26
  RESLIB, 12–27
  RESSUP, 12–29
  RNDSEG, 12–30
  STACK, 12–31
  (SUPLIB), 12–32
  SVDB$, 12–24
  SVTK$, 12–34
  TASK, 12–33
  TSKV, 12–34
  UNITS, 12–35
  VSECT, 7–14, 12–38
  WNDWS, 12–39
Options, 2–10, 10–3, 12–1 to 12–39
  summary, 12–1 to 12–2
Ordering program sections, 11–25
$OTSV, C–9
OUTS PC instruction, 7–9
Overlay Description Language, 3–1, 3–3 to 3–6, 13–1 to 13–5
Overlay Map (/MP), 11–18
Overlays
  brief discussion, 1–3
  co-trees, 4–1 to 4–17
  data structure, C–11
  definition, 3–2
  description of memory map, 3–7e
  designing, 3–7
  for COBOL programs, 3–2
  memory-resident, 7–4 to 7–7, 11–23
  memory-resident, in cluster libraries, 7–8
  ODL file, 11–18
  simple, 3–1 to 3–15
  using the /MP switch, 11–18
Overlay tree, 3–10
OVR attribute, 6–4, 6–5, 12–12, 13–4

P

Parentheses
  nesting, 3–9
  ODL operators, 3–4, 13–4
PAR option, 7–3 to 7–4, 12–25
Partition, 7–3 to 7–4
Partition for Resident Area (PAR), 12–25
Patching, 12–4
  offset from global, 12–17
Path (overlay structure), 3–10 to 3–13

PDP-11 C, 1–1
  run-time system for, 2–2
PDP-11 COBOL, 1–1
  disk libraries for, 2–4t
  example build, 2–16
  run-time system for, 2–2
Performance, improving TKB, E–1 to E–3
Physical memory, 2–2
PIC
  See *Position independent code*
/PI switch, 7–3, 11–21
PMDUMP, 11–22
/PM switch, 11–22
Position-independent (/PI), 11–21
Position independent code, 7–2 to 7–3
  and cluster libraries, 7–8
Post-mortem Dump (/PM), 11–22
Programmer Control Regions, 7–26 to 7–28
Program Name for SYSTAT (TASK), 12–33
Program sections, 6–1 to 6–15
  absolute, 13–4
  allocating space for global, 6–2
  appearing in map, 6–6
  attributes, 6–1 to 6–4, 13–3
  changing order of, 11–33
  concatenated, 6–4, 13–4
  data, 6–4
  definition, 6–1
  extending size of, 12–12
  global, 6–2, 13–4
  in default library, 4–17
  instruction, 6–4
  in SYSLIB.OLB, 3–15
  local, 13–4
  overlaid, 6–4, 13–4
  placing with .PSECT, 6–5
  read/write, 6–2 to 6–4, 11–19, 11–33, 13–4
  read-only, 6–2 to 6–4, 11–19, 11–33, 13–4
  relocatable, 13–4
Program size, 2–2, 3–1
Program status word, 11–36
Program version ID, B–10
Project-programmer numbers, specifying, 2–9
.PSECT, 6–5, 13–3
PSECT, B–5 to B–9
  additive displaced relocation, B–23
  additive relocation, B–22
  displaced relocation, B–21
  item type, B–32
  relocation, B–21
  reserved names, D–3 to D–5
  with I- and D-space tasks, 8–1 to 8–4
.PSECT directive (MACRO), 6–1
PSW, 11–36

R

R$LDAT, C–6
R$LFLG, C–6
R$LHGV, C–6
R$LLDZ, C–6
R$LMXV, C–6
R$LMXZ, C–6
R$LNAM, C–6
R$LOFF, C–6
R$LSA, C–6
R$LSEG, C–6
R$LWND, C–6
Read/write resident library, 2–11
Read-only resident library, 2–11
Record Management Services, 2–4t, 2–6
Region descriptor, C–19
Relative addressing, 1–3, 7–2
REL attribute, 6–5, 13–4
Relocatable/Relocated records, B–30
Relocation
  additive, B–26
  complex, B–24
  directory format, B–14f
  entry, B–15
  global, B–16
  global additive, B–18
  global additive displaced, B–18
  global displaced, B–17
  internal, B–15
  internal displaced, B–16
  PSECT, B–21
  PSECT additive, B–22
  PSECT additive displaced, B–23
  PSECT displaced, B–21
Relocation directory, B–14
RESCOM option, 12–26
Reserved symbols, D–1 to D–3
Resident area, 7–1 to 7–7, 11–21, 12–10, 12–25
  absolute, 7–2, 7–3
  position independent, 7–2 to 7–3
Resident common, 12–10, 12–26
  building your own, 7–1
  definition, 7–1
Resident libraries, 2–5 to 2–7, 12–21, 12–27
  building your own, 7–1
  clustering, 2–11
  definition, 7–1
  limit in a cluster, 12–7
  maximum number, 2–7, 2–10
  read/write, 2–11
  read-only, 2–11
  system-owned, 2–10
  user-owned, 2–11
Resident Overlay (/RO), 11–23
Resident Supervisor-Mode Library, 12–29
RESLIB option, 2–10 to 2–11, 12–27
  example, 2–17
RESSUP option, 12–29
Revectoring cluster libraries, 7–10
RLD record, B–1
RMS, 2–4t
RMS-11 libraries in a cluster, 12–7
RMSDAP.OLB, 2–4t
RMSLIB.OLB, 2–4t
RMS resident libraries, 2–6
RMSRES library, 2–12, 12–7
RNDSEG Option, 12–30
RO attribute, 2–11, 6–4, 13–4
  in CLSTR option, 12–8
  in cluster libraries, 2–14
Root, 3–2, 3–8
  co-tree structure, 4–1
  null for co-tree, 4–4
  putting libraries at end of, 3–13
  simple overlay structure, 3–10
.ROOT, 13–4
.ROOT command, 3–3 to 3–5
/RO switch, 11–23

Round Segment (RNDSEG), 12–30
Routines
  library (in co-trees), 4–6
  library (in simple overlays), 3–13
RSX run-time system, 2–11
$$RTS, 6–7
Run, diagnostic, 10–2
Running the Task Builder, 2–7, 10–1 to 10–6
Run-time system, 2–2 to 2–3
  RSX, 2–11
RW attribute, 2–11, 6–4, 6–5, 13–4
  in CLSTR option, 12–8
  in cluster libraries, 2–14

S

Sample program
  first build, 4–9
  second build, 4–10
  third build, 4–15
  using co-trees, 4–6
SAV attribute, 13–4
/SB switch, 11–24
Segment
  as described in map, 6–6
  definition, 3–10, 5–4
  descriptor, C–14
  linkage, C–15
  overlay format, C–19
  putting libraries at end of, 3–13
  root, 3–8
  root format, C–19
Segmentation facility, 3–2
Segregate Program Sections (/SG), 11–25
Selective Search (/SS), 11–34
Sequential (/SQ), 11–33
Set SST Vector Table for Debugging Aid (SVDB$), 12–24
Set SST Vector Table for Task (SVTK$), 12–34
/SG switch, 11–25
Sharable code, 2–7
Short Map (/SH), 11–26
/SH switch, 6–6, 11–26
Single slash (/), to end command line, 10–3
Slow build, 11–24
SMRES library, 2–12, 12–7
Spool Map Output (/SP), 11–32
/SP switch, 11–32
/SQ switch, 11–33
/SS switch, 11–34
SST vector, 12–24, 12–34
Stack, 12–31
  changing size, 6–7
  definition, 6–7
  for memory-resident areas, 7–2
STACK option, 7–2, 12–31
Start-of-segment item type, B–28
.STB file, 2–8, 2–11, 7–2, 12–20, B–26, B–29
Supervisor-Mode Library, 9–1 to 9–22, 12–32
SUPLIB Option, 12–32
SVDB$ option, 12–24
SVTK$ option, 12–34
Switch
  /CC, 11–3
  /CO, 11–4
  /DA, 11–5
  /DL, 11–6
Switch (Cont.)
  /EL, 11–7
  /FM, 11–8
  /FO, 11–9
  /FP, 11–10
  /FU, 11–11
  /HD, 11–12
  /ID, 11–13
  /LB, 11–14
  /LI, 11–16
  /MA, 11–17
  /MP, 11–18
  /MU, 11–19
  /NM, 11–20
  /PI, 11–21
  /PM, 11–22
  /RO, 11–23
  /SB, 11–24
  /SG, 11–25
  /SH, 11–26
  /SP, 11–32
  /SQ, 11–33
  /SS, 11–34
  /TR, 11–36
  /WI, 11–37
  /XT, 11–38
Switches, 11–1 to 11–38
  overview, 11–1 to 11–2
Symbolic debugger, 2–8
Symbols
  ambiguously defined, 3–12, 4–6
  global, 3–11
  local, 3–11
  multiply defined, 3–12, 4–6
  reserved, D–1 to D–3
  undefined, 3–12, 4–6, 4–17
Symbol table, Task Builder's internal, 11–34
Symbol table file, 2–8, 7–2, 10–1, 12–20
Synchronous system trap, 12–34
SYSLIB.OLB, 2–4t, 3–15, 11–6, 11–17, 11–33, 11–36
SYSTAT, 12–33
System Common Block (COMMON), 12–10
System default library, 3–15, 11–6, 11–11, 11–17
  how searched for co-trees, 4–3
  using co-tree techniques on, 4–17
System-owned resident library, 2–10

T

Table
  jump, 7–10
  vector, 7–10
Task, extending memory for, 12–13
Task Builder
  aborting run, 12–3
  command line, 2–8, 10–1
  data formats, B–1
  exit on error, 11–38
  improving performance, E–1 to E–3
  options, 12–1 to 12–39
  running, 10–1 to 10–6
  switches, 11–1 to 11–38
  work file, E–1
Task file, 2–8, 7–2, 10–1
Task identification item type, B–29
TASK option, 12–33
Task SST Vector (TSKV), 12–34

T-bit, 11–36
Text information record format, B–12f
Time, reducing disk access, 3–9 to 3–10
TKB-generated record, B–28
Trace, 11–36
TRACE.OBJ, 11–36
Traceable Program (/TR), 11–36
Transfer address, B–6
Trap, synchronous, 12–34
Tree
  co-tree structure, 4–1
  simple overlays, 3–10
/TR switch, 11–36
.TSK file, 2–8, 2–11, 7–2
TSKV option, 12–34
TXT record, B–1

U

Undefined symbols, 4–17
  in a simple overlay, 3–12
  in co-trees, 4–6
UNITS option, 12–35
  example, 2–15
User-owned resident library, 2–11
UTILITY, 7–1, 11–19

V

Vector
  autoload indicator, 3–4
  definition of autoload, 5–1
  extension area, C–9, C–10f
  revectoring cluster libraries, 7–10
  SST, 12–24, 12–34
  table, 7–10
  table code sample, 7–12
$VEXT, C–9
Virtual address space, 2–2
Virtual Program Section (VSECT), 12–38
Virtual Program Sections, 7–14 to 7–26
VSECT option, 7–14, 12–38

W

Wide Listing Format (/WI), 11–37
Window descriptor, C–18
Windows, declaring maximum number of, 12–39
/WI switch, 6–6, 11–37
WNDWS option, 12–39
Work file, E–1

X

XRB, 11–12
/XT switch, 11–38

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

RSTS/E Task Builder Reference Manual
AA–5072D–TC

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

Do Not Tear - Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST- CLASS MAIL PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
CONTINENTAL BOULEVARD MKO1-2/E12
PO BOX 9501
MERRIMACK NH 03054-9982

Do Not Tear - Fold Here

Cut Along Dotted Line

Reader's Comments

RSTS/E Task Builder Reference Manual
AA–5072D–TC

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

Do Not Tear - Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST- CLASS MAIL PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
CONTINENTAL BOULEVARD MKO1-2/E12
PO BOX 9501
MERRIMACK NH 03054-9982

Do Not Tear - Fold Here

Cut Along Dotted Line

digital

Printed in U.S.A.